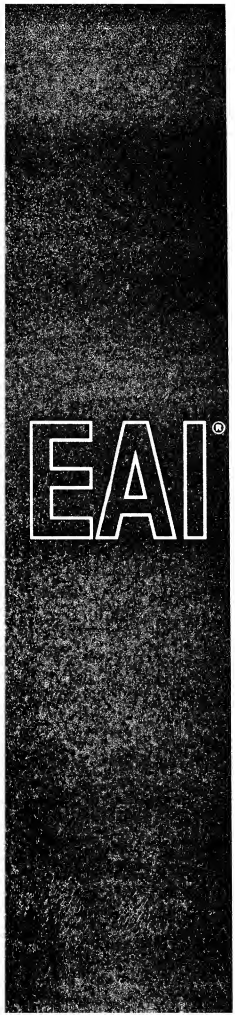


Prepared for
GEORGE C. MARSHALL SPACE FLIGHT
CENTER, NASA
Huntsville, Alabama

STORED PROGRAM CONCEPT FOR ANALOG COMPUTERS

FINAL REPORT

EAI PROJECT 320009
NASA ORDER NAS8-21228

PRINTED IN U.S.A. JUNE 1968

STORED PROGRAM CONCEPT FOR ANALOG COMPUTERS

Final Report

Prepared for

GEORGE C. MARSHALL SPACE FLIGHT
CENTER, NASA

Huntsville, Alabama

EAI Project #320009

NASA Order #NAS8-21228

George Hannauer
Applications Engineer
Digital/Hybrid Programming
Department

Electronic Associates, Inc.
Princeton, N.J.

TABLE OF CONTENTS

1. INTRODUCTION

Eliminating the patchpanel has long been the dream of analog computer users, for a number of reasons. One of these is psychological; the sight of a large mass of wires hanging from the front of a computer is ugly and disturbing to almost anyone except a true "dyed-in-the-wool analog man", and even some members of this select group are more than willing to see the patchpanel give way to a deck of cards or a punched tape. These aesthetic considerations alone are enough to discourage many novices and cause them to turn to a digital computer with a simulation language.

Aesthetics aside, there are sounder reasons for desiring to replace the patchpanel with a switching matrix. The most obvious among these are the following:

a. Ease of Programming. Patching an analog problem is tedious and time-consuming, and there is little doubt that an automatic patching system (in conjunction with appropriate software) would greatly simplify programming. This is desirable, not because programmers are lazy, but because they are expensive. The cliche "time is money" is as applicable to the analog computer (and programmer) as it is to the digital.

b. Stored-Program Capability. Patchpanels can be stored, of course, but good ones are expensive, heavy, and space-consuming. Cards and tape provide a much more desirable storage medium.

c. Reliability. This is something of a question mark, since it is not clear that a relay matrix, static card reader, or other system would be inherently more reliable than a patchpanel. However, there is some reason to believe that it would be, if only because it is stationary. Most of the wires that fall out of present-day patchpanels do so because the panel must be carried away from the machine for storage.

In addition to the above considerations, an additional application is conceivable, namely, some form of multi-programming. By this I do not mean allowing two people to share the same components in alternate 10-millisecond time intervals, but rather allowing one user to use 75% of the components in a machine for a big problem while another user uses the remaining 25% for a smaller problem. This is possible in principle on a patchpanel, and, in fact, has been done at EAI's Princeton facility, but is not very practical; the users get in each other's way.

It is possible that an automatic programming/patching system would have the same effect on analog computer utilization that FORTRAN had for the digital computer. IBM is currently publicizing the fact that before FORTRAN was developed, a leading business magazine predicted that the total market for digital computers (not the annual market) would be limited to about fifty consoles, because the programming was so involved and expensive! Since then, the total number of installations has exceeded this estimate many times over. If the demand for analogs were to expand by the same order of magnitude, the market would be a healthy one indeed.

Any automatic patching system will obviously be expensive. Previous attempts to design such systems for medium-to-large systems (200 to 400 amplifiers and up) have run into "the N^2 problem", that is, the fact that the number of switches necessary grows faster than linearly with matrix size. Keeping the total size

and cost of such a system within reasonable bounds requires a two-pronged attack on the problem: reducing the number of necessary switches to a minimum, and designing a switch with sufficiently low cost, noise, contact resistance, and crosstalk.

In 1967, NASA contracted with EAI for a feasibility study for the development of an automated patching system. This report covers the results of phase 1 of that study: the reduction of the number of switches.

The last study made of this problem within EAI was done by Joe Marshall and Bill Hagerbaumer in 1961. Concurrently, Wolfgang Ocker undertook a similar study for a DDA. The 231-R analog used by Marshall and Hagerbaumer as the basis of their study was approximately equivalent to the 680 used in this study. Marshall and Hagerbaumer concluded that between 25,000 and 35,000 switches would be necessary. However, they did not actually test their proposed system on any actual problems. Examining their proposed system in the light of actual problems, it appears that the number of switches required would be somewhere between 50,000 and 100,000.

At the start of this project, I was given a target value of "under 10,000 switches for a 680-size machine". Thus the task is to effect a reduction by a factor of five to ten.

This report describes two proposed systems: a "small" system (one 680) requiring between 8,000 and 9,000 switches, and a "large" system (two 680"s or an 8800) requiring between 21,000 and 22,000 switches. Details of the design are given in Chapters 6 and 7. Furthermore, the switching system appears adequate to handle typical analog programs, including those furnished by NASA as sample programs to implement (see Chapters 8 and 9). Thus the design goals have been met.

It should be pointed out that the "typical 680" used in this report is larger than the fully-expanded standard 680: it contains 30 integrators, 36 summers, 3 resolvers, 39 multipliers (not counting the 12 multipliers within the resolvers) and 18 variable DFG's. It is possible to get all this equipment into one system consisting of a 680 console and half of a resolver expansion rack. Hence the large system (60 integrators, 72 summers, 6 resolvers, 78 multipliers, 36 variable DFG's) would fit into two 680 consoles and one resolver expansion rack.

This raises the question of whether the system should be implemented on an existing analog (by hard-wiring the relay matrix to a patchpanel) or whether a new machine should be developed. The system is obviously "cleaner" and more economical if the analog is designed from the ground up to accomodate the switching matrix, but obviously, any prototype matrix must be evaluated on an existing machine. Furthermore, the possibility of producing a plug-in matrix for existing machines as a standard product should not be overlooked.

I have kept the 680 in mind throughout this project as a possibility for plug-in implementation of the matrix. This is in accord with the wishes of NASA personnel and also with my own inclinations, since I am more familiar with this machine than any other. The resulting design appears feasible for implementation on a 680 (or 8800) with two qualifications. One of these is minor; the other may be major.

The minor qualification has to do with configuration switches (that is, the switches that convert a summer into an integrator or a multiplier into a divider). Ideally, these switches should be within the analog console itself, for two reasons: first, because they do not involve component-to-component connection (and hence do not need to be in the main matrix), and second, because some of them switch summing junctions, and hence should be kept as close as possible to the components themselves. There is a third reason: some of them involve form-C or multi-pole switching and hence might be more economical with conventional relays. To implement the configuration switching on a 680 or 8800 may require some changes in internal tray wiring, but I do not believe this problem is insurmountable.

The more serious problem is the handling of pots. To keep the number of inputs and outputs down (and to avoid the switching of relatively high-impedance pot outputs) pots are committed to summer inputs, integrator inputs, integrator IC's, and comparator biasing. This, of course, increases the necessary pot complement, since in any given problem, some of the pots will not be used. I strongly believe that the number of contacts saved justifies this decision (further information on this point is given in Chapter 6), and there is no inherent problem in increasing the pot complement in future machines if this decision is, in fact, sound. However, it raises the question of what to do with existing machines. The 680 configuration used in this report requires over 300 pots, so that implementing the full system on an existing 680 would require a pot expansion rack. The pots are the only components (except for the configuration relays and the switching matrix itself) that need to be added to the 680 to complete the system.

As far as a prototype is concerned, the pots should be no problem, since a prototype (for evaluating noise, crosstalk, and packaging problems) need not be full-size. There should be enough pots on a fully-expanded 680 for implementing a prototype matrix.

The proposed design divides the 680 into six nearly-identical modules. Components within a module are interconnected by means of a small matrix, and the components have limited access to inputs and outputs of other modules through a large matrix. Hence, for a prototype, it is only necessary to build a matrix for about one-third of the machine (two modules). This will allow the concept and the hardware to be evaluated without any additional external analog equipment.

2. DEFINITION OF TERMS

This section defines most of the terminology to be used in this report. It is divided into sub-sections to facilitate reference.

2.1 Matrix, inputs, outputs

A matrix is any collection of switches that allows terminals of one type (called inputs) to be connected to terminals of another type (called outputs). The use of the terms "input" and "output" requires some clarification; the inputs to the switching matrix are the outputs of computing components and vice-versa. Thus, whether a given terminal is called an input or an output depends on whether one looks at it from the viewpoint of the computing components or from the viewpoint of the switching matrix. Wherever there might be doubt, the term "matrix input", "component output", etc. will be used. When used alone, the terms "input" and "output" will be used from the viewpoint of the matrix; that is, the term "input" will, by convention, refer to a matrix input (a component output).

2.2 M and N, Expansion Factor E

The letters N and M will be used to refer to the number of matrix inputs and outputs respectively. In many cases, it is desirable to describe the size of the matrix in terms of a single parameter, rather than two. For this reason, we define the expansion factor of a matrix by $E = M/N$. The matrix may then be described by specifying the size by means of the parameter N and the "shape" by the parameter E. Note that when two or more identical computers (each with N matrix inputs and M matrix outputs) are slaved together, the result is a larger computer with the same expansion factor. The expansion factor is thus independent of size. It depends on the computer configuration. A preliminary investigation indicates that E will lie in the range from 1.5 to 2.5 for a practical analog programming matrix.

2.3 Expander, concentrator, simple matrix, composite matrix

A matrix with more outputs than inputs ($E>1$) will be called an expander. A matrix for which $E<1$ will be called a concentrator. If $E = 1$, the matrix is a square matrix. The term "concentrator" is taken from telephone usage; the term "expander" is a natural extension.

Matrices may be classified according to the number of switching stages; an n-stage matrix is one in which every path from an input to an output

passes through n switches. If some paths take more switches than others, the matrix is a composite matrix; if all paths are of the same length, it is a simple matrix. The telephone company uses composite matrices (a long-distance call uses more switches than a local call) and it seems reasonable to consider the same sort of approach for analog programs.

2.4 Rectangular Matrices

A one-stage matrix is, by definition, a matrix in which each input-to-output connection passes through a single switch. For N inputs and M outputs, such a matrix is usually arranged (physically and in circuit diagrams) in an N x M rectangular array, and hence is often called a rectangular matrix. If any input is to be connectable to any output, then MN switches are required. Such a matrix will be called a complete rectangular matrix. A rectangular matrix with fewer than MN switches will be called an incomplete or restricted rectangular matrix. Some restriction of matricies appears desirable (for example, a pot output need not be connected to the input of another pot, nor to its own input, nor to the inputs on a multiplier).

Note, in passing, that if the number of switches is expressed in terms of the parameters N and E, a complete rectangular matrix requires N^2E switches. This gives the expected result that for rectangular matricies of different sizes but with the same value of E, the number of switches is proportional to the square of the matrix size. A straightforward rectangular matrix requires far too many switches to be practical, but it serves as a natural basis of comparison for evaulating other matrix configurations.

2.5 Programs

The purpose of any GPAC switching matrix is to implement programs. From the matrix point of view, a program may be defined as a list of connection statements of the form "connect input i to outputj" where $1 \leq i \leq N$ and $1 \leq j \leq M$. However, not every such list represents a valid analog program. We must add an additional restriction that no two connection statements in a program involve the same matrix output. Such a pair of connection statements would require that two analog component outputs be tied to the same input (remember component outputs are matrix inputs and vice versa). It is, however, perfectly possible for a single matrix input to connect to many different outputs; this corresponds to an analog component whose output drives several other components. A matrix input that is to be conncected to n different matrix outputs is said to have a fanout of n .

Since a matrix output connects to at most one matrix input, an alternative mathematical description of a program is as a single-valued function whose domain is the set of matrix outputs and whose range is the set

of matrix inputs plus one additional symbol which represents "no connection". In other words, one may index the N inputs 1, 2, ... N, with the symbol "O" representing "no connection" and define for each output j ($1 \leq j \leq M$) the function

$f(j) =$ i if matrix output j is connected to input i

$f(j) =$ o if matrix output j is not used.

This approach makes it immediately obvious that the number of possible programs for an N by M matrix is $(N+1)^M$. If the matrix contains n switches, then it has 2^n possible states, and hence if it is to be able to handle all possible programs, we must have $n \geq M \log_2(N+1)$.

In fact, if N is one less than an integral power of 2, say $N = 2^k - 1$, then it is possible to design a switching matrix with $M \log_2(N+1) = Mk$ switches which can handle all programs. The catch is that each "switch" must be a relay with several sets of contacts. Each matrix output is wired to the arm of a form- C relay, and each contact of this relay is wired to one of the arms of a 2 pole relay (2 form C sets of contacts), and so on. This "binary tree" design uses k relays per matrix output (Mk relays total) and allows each matrix output to connect to any one of the $N = 2^k - 1$ inputs, or to nothing (the last matrix input being grounded).

However, since the first input stage must be a relay with $2^k - 1$ form C contacts, the size of the relays makes this scheme impractical. The technique might prove practical with cascaded stepping switches rather than relays, but the cost would probably be prohibitive.

Incidentally, if this scheme is implemented with relays, then since each relay has twice as many contacts as the preceding one, the total number of sets of form -C contacts per matrix output is $1+2+4+\ldots+2^{k-1}$, which sums to $2^k - 1 = N$. Hence the total number of form C contacts for the entire N by M matrix is MN. If each set of form-C contacts is replaced by two form-A contacts, this binary tree method takes exactly twice as many form A switches as a simple rectangular matrix.

2.6 Abstract and Concrete Programs

The definition of a "program" in the previous section leaves out one important point; the interchangeability of components. The analog programmer does not (or, at any rate, should not) care which components of a given type are used to solve a given problem. If integrator 35 and integrator 45 are interchanged, the program (in the sense of the connection statements) is changed, but the same analog problem is being solved.

To keep this distinction in mind, let us define a concrete program as a set of connection statements (or, equivalently a function from outputs to inputs) in the sense of the last section. Two concrete programs will be called equivalent if one is obtainable from the other by permutation of components of similar type (i.e. renumbering the integrators, the pots, the summers, etc.) An abstract program is defined as an equivalence class of concrete programs. In other words we "identify" any two programs which are "abstractly equivalent". An abstract program is essentially what the analog programmer has produced after he has drawn the circuit diagram, complete with all component interconnections, but has not written any numbers inside the component symbols. After assigning all components, he has reduced the abstract program to a concrete program.

A digital computer routine for implementing a program should probably consist of two successive algorithms: a component assignment algorithm for assigning components (thus reducing the abstract program to a concrete one), followed by a switch assignment algorithm, which chooses appropriate switch paths to implement the concrete program. Initially, it appeared desirable to combine the two: that is, at each stage of the algorithm, choose the "nearest" component of the desired type and the choose the best switch path to implement the desired connection. The result would be a single algorithm assigning components and switch paths alternately. This line of attack has proved fruitless. The two types of choices (choice of component and choice of switch path) seem too dissimilar to "mesh" well. It now appears that two separate algorithms are preferable (although if the switch-assignment algorithm experiences blocking, it may be possible to re-assign a few components to relieve the blocking and try again.

3. COMPARISON WITH THE TELEPHONE SYSTEM

Any study of the automatic patching problemmust, at some point, make reference to the experience of the Bell System. This is so for two reasons. The most obvious reason is the success of the system in getting around "the N^2 problem", that is, the fact that the number of switches in any switching matrix grows faster than linearly with the matrix size.

A system of over ten million telephones, which allows (with rare exceptions) anyone to call anyone else must have a large number of switches. If a straight - forward rectangular array were used, ten million phones would require 10^{14} switches. Even at the ridiculously low price of a penny a switch, such a system would cost 10^{12} dollars, which is more than the entire country's annual Gross National Product. Even more interesting is the marginal cost of adding one additional subscriber. If this subscriber is to have direct access to all ten million existing subscribers, ten million additional switches would be required. Using again the optimistic price of a penny a switch, the subscriber would have to pay 100,000 dollars for the installation.

Now, of course, the Bell System does not provide direct access between subscribers, but instead provides indirect access through several levels of switching. It is the multi-stage nature of the switching system that prevents the number of switches from growing excessively. In fact, as the number of telephones has increased, the cost per telephone has actually decreased - a remarkable tribute to the efficiency of the system.

The second reason for interest in the Bell System is that much of the published literature on switching theory is based upon the particular switching problem faced by Bell. In fact, the three-stage matrix theory presented in Chapter 4 is based on ideas originally published by Charles Clos in the Bell System Technical Journal.

Despite the wealth of published literature on the subject of switching theory, there is very little that is relevant to the automatic analog patching problem. This is because our problem differs from Bell's in a number of important respects. It is worthwhile to list these differences to obtain some perspective on the nature of the automatic patching problem. There are basically six significant differences, three of which tend to make our problem easier than Bell's and three of which tend to make it harder. Taken together, they make the problem not necessarily harder or easier than Bell's, but merely different.

3.1 Size

The largest contemplated analog system is much smaller than the Bell System. This fact not only tends to make our problem easier than Bell's, but changes its character as well. In Chapters 4 and 5, we will see that a simple rectangular matrix is the most economical design for small systems, while a large system is more efficient with multi-stage matrices. Thus differences in size mean differences in type as well.

3.2 Static Versus Dynamic Considerations

The Bell System must allow dynamic access between callers, that is, a switch path must be chosen for a given connection without prior knowledge of what other connections may be made in the future. Confronted with a Chicago-San Francisco call, the system has to decide whether to route it through Denver or Salt Lake City without knowing which city is going to have the heaviest traffic a few minutes later. In contrast, the analog programmer knows all connections before the first connection

is made, and once made, they do not change. Occasional program changes (the equivalent of re-patching) are made so seldom that the entire switch assignment may be re-computed if necessary. Hence our problem is static, while Bell's is dynamic. In the static problem, every switch path may be chosen with full knowledge of the entire problem. In Chapter 4, we will see that this fact "buys" us about a factor of two; that is, the static case takes about half as many switches as the dynamic case.

3.3 Interchangeability

In analog programming it is not necessary for every component to be able to communicate with every other component, since component assignment is arbitrary. If I want to connect a summer to an integrator, I may not care whether the summer is A21 or A36, but when I want to talk to Joe Green in Chicago, I won't be satisfied with Jim Black in Cleveland instead. Summers are interchangeable; integrators are interchangeable; people aren't. In other words, we are interested in abstract, rather than concrete programs (See Chapter 2). The ability to interchange components at will should prove a significant advantage, if we take advantage of it properly.

The preceding three points are all in our favor. The three disadvantages are as follows:

3.4 Traffic Density

At any given time, only a small percentage of the country's telephones are in use. In contrast, a typical analog program may use most of the components in the machine. Hence, although Bell works with much larger matrices, the traffic on them is much lighter. One way that Bell takes advantage of this fact is to use concentrator matrices at the inputs to their large switching systems. This keeps the number of inputs and outputs to their large systems within reasonable bounds, but it also means that only a small fraction of the people in a given exchange can use the phone at any given time, and a still smaller fraction may make long-distance calls. Fortunately, traffic is normally light enough that blocking does not often take place. In the event of a catastrophe, such as a fire, flood, or earthquake, when many people in a given area want to use the phone at once, a number of them will find the trunklines all busy. The analog programmer must operate under such conditions of moderate-to-heavy traffic density most of the time.

3.5 Fanout

Fanout is quite rare in the Bell System, but quite common in analog programs. Cases of fanout occuring within the Bell System (e.g. conference calls) probably require manual intervention; at any rate, I don't know any way to set up a conference call between three different cities by direct dialing.

The "cost" of fanout in terms of switches is less than I had originally anticipated; comparison of the results of Chapter 4 and Chapter 5 indicate that the cost is about $\sqrt{2}$; that is, about 1.4 times as many switches are needed to cover the case of fanout.

3.6 Blocking Penalty

In order to program an analog problem it is necessary to make all connections; if even one connection is omitted, the entire simulation is affected. A typical 200-amplifier problem may require 500 connections. If one of these is blocked (and cannot be un-blocked by re-assigning components) then the problem can not be run. If this happens in the Bell System, it merely means that one person in 500

can't complete his call because the trunk lines are busy; the other 499 people are satisfied and the 500th can always try again later. (In this case, the dynamic nature of the Bell System pointed out in section 2 becomes an advantage). Thus a level of blocking that Bell might find acceptable may be unacceptable in an analog switching scheme.

The advantages that the analog programmer enjoys over the Bell System may be summarized as follows: he is concerned with a relatively small switching matrix making static connections between interchangeable components, while Bell's System is large, and makes connections dynamically between unique subscribers. Bell's advantages may be summed up as follows: they operate under conditions of low traffic density (low percentage utilization) and negligible fanout, and they can tolerate a small percentage of blocked connections at any time. The analog programmer operates under conditions of high traffic density (over 50% of the components are in use in a given problem) and moderate-to-heavy fanout (fanouts of 2 or 3 are quite common, and fanouts of 8 to 10 occur occasionally). Furthermore, any blocking is not merely undesirable; it is catastrophic.

4. THREE STAGE MATRICES (WITHOUT FANOUT)

The theory of the three-stage matrix is a natural starting point for any study of automated analog patching. Almost all proposed systems utilize three-stage matrices (sometimes in combination with other matrices). If fanout is ignored, the theory assumes an especially simple form. It is fairly easy to prove that a three-stage matrix can be found that can handle any non-fanout program with fewer switches than a straightforward rectangular matrix, if the matrix is large enough. Most of the theory in this chapter is due to Clos [1], but is summarized here in a slightly different form to facilitate generalization to the fanout case, which is covered in Chapter 5.

4.1 Parameters of the three-stage Matrix

The type of three-stage matrix considered in this report consists of three types of rectangular matrices, called input blocks, middle blocks, and output blocks. The input blocks are connected to the matrix inputs, the output blocks are connected to the matrix outputs, and the input blocks are connected to the output blocks through the middle blocks. Each middle block has one connection to each input block and one connection to each output block. Figure 4.1 illustrates the configuration. The input blocks are 2 by 2 rectangular matrices; the middle blocks are 3 by 4 rectangular matrices, and the output blocks are 2 by 3.

An arbitrary matrix of this type is characterized by the following parameters:

N	the number of inputs
M	the number of outputs
n	the number of inputs per input block
m	the number of outputs per output block
X	the number of input blocks
Y	the number of middle blocks
Z	the number of output blocks

From the definition, it is obvious that $N = Xn$ and $M = Zm$. Eq. 4.1

The total number of switches in the matrix may be expressed in terms of the parameters as follows:

The input blocks are n by Y matrices; hence they each contain nY switches. Hence the total number of switches in all input blocks is XnY. Since $Xn = N$, this can be written as NY. The same reasoning leads to the conclusion that MY switches are needed in the output blocks. Finally, the middle blocks have X inputs and Z outputs each, and hence the total number of switches required in the middle blocks is XYZ. Thus the total number of switches is given by

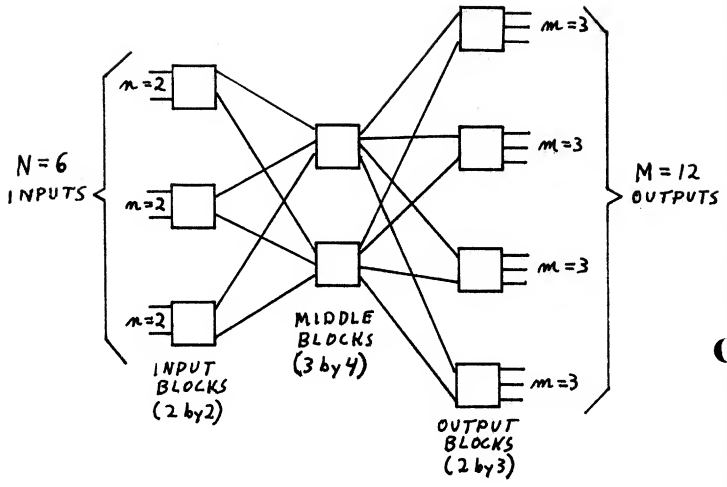

N = 6
INPUTS
n = 2
n = 2
n = 2
INPUT
BLOCKS
(2 by 2)
MIDDLE
BLOCKS
(3 by 4)
m = 3
m = 3
m = 3
m = 3
OUTPUT
BLOCKS
(2 by 3)
M = 12
OUTPUTS

Figure 4.1

If we eliminate X and Z in favor of m and n, we get a form that turns out to be more useful:

$$S = Y\,(M+N+\frac{M}{m}\cdot\frac{N}{n}) \qquad \text{Eq. 4.2}$$

In a practical design problem, M and N may be taken as given (that is, the problem is to find the most efficient matrix for connecting N inputs to M outputs). Hence m, n, and Y are the design parameters to optimize. We want the minimum number of switches for an adequate matrix, where the term "adequate" means that the matrix contains enough switches to implement a particular class of programs. Any definition of "adequate" will impose some constraints on m, n, and Y and the problem becomes one of choosing m, n, and Y to minimize S subject to these constraints. Since S is a monotonic increasing function of Y and a monotonic decreasing function of m and n, we want to choose Y as small as possible and m and n as large as possible. Thus one would expect any definition of "adequacy" to impose a lower bound on Y and /or upper bounds on m and n.

In this chapter, two definitions of "adequate" are considered (for the case of non-fanout programs). In the next chapter, an additional definition is considered for the case of fanout. In all cases, the definitions lead to upper bounds on m and n and lower bounds on Y, as expected. The problem thus reduces to a problem in constrained parameter optimization. Solution of these problems yields design criteria for optimal three-stage matrices and formulas for the optimal numbers of switches.

4.2 Static and Dynamic Adequacy

A matrix will be called statically adequate if it is capable of handling any concrete program (that is, any concrete program, as defined in Chapter 2 can be implemented on it). It will be called dynamically adequate if any transition from one such program to another can be made without breaking any connections common to both. The dynamic case is not particularly important for an analog programming matrix, since transitions from one program to another (the equivalent of re-patching) are normally made between runs, not during runs. Switching during runs will ordinarily be done with electronic switches for reasons of speed. The dynamic case is, of course, the one that interests the telephone company, and hence most of the published theory is based on dynamic considerations.

The terms "statically adequate without fanout" and "dynamically adequate without fanout" will be used in the obvious sense (one considers only non-fanout programs). In the non-fanout case, the adequacy conditions assume very simple forms as follows:

Therorem 4.1 For a three-stage matrix to be dynamically adequate without fanout, it is necessary and sufficient that $Y \geq m+n-1$.

Proof: At any given time, it must be possible to connect any unused input to any unused output. There are n-1 other inputs on the same input block as the input to be connected. If these inputs are all in use, they tie up n-1 middle blocks. (No middle block can handle more than one connection from a given input block). Similarly, the m-1 other outputs on the output block might all be in use, tying up an additional m-1 middle blocks. Thus at most m+n-2 middle blocks are tied up by existing connections. The actual number of middle blocks tied up may be less than this, because some of the n-1 middle blocks that are tied up on the input side may be the same as some of the m-1 middle blocks tied up on the output side. However, the worst case occurs when there is no such overlap (and there will be no overlap if there is no pre-existing connection between the input block and the output block under discussion). In this case m+n-2 middle blocks are tied up, and if $Y \geq m+n-1$, there will be one middle block left to make the necessary connection. Since this is a "worst case" the condition is both necessary and sufficient.

Therorem 4.2 For a three-stage matrix to be statically adequate without fanout, it is necessary that $Y \geq m$ and $Y \geq n$.

Proof: In order to handle the case where all m outputs in a given output block are in use, we must have $Y \geq m$. In order to handle the case where all n inputs in a given input block are in use, we must have $Y \geq n$. Hence both conditions are necessary.

Furthermore, it appears likely that these two conditions taken together are also sufficient, that is, that any non-fanout program can be implemented on a three-stage matrix as long as the number of middle blocks is at least as great as m and n. This is asserted without proof in the literature [2]. I suspect it could be proved rigorously, but the proof appears trickier than I had first suspected. At any rate, practical experience indicates that it is at least approximately sufficient; that is, that the number of middle blocks is necessary, at worst, not much greater than m or n. For the purposes of this report, it will simply be assumed that these conditions are sufficient. If this assumption turns out to be false, it will not really affect the overall conclusions of this report, since the

real practical interest is in programs with fanout. The non-fanout case is of interest only as a prelude to the more realistic case covered in Chapter 5.

With these assumptions, the problem becomes one of minimizing S (given in Eq. 4.2) subject to the constraints

$$Y \geq m+n-1 \quad \text{for the dynamic case} \qquad 4.3$$

$$Y \geq m \text{ and } Y \geq n \quad \text{for the static case} \qquad 4.4$$

4.3 Optimizing the Matrix

As expected, the constraints impose upper bounds on m and n and lower bounds on Y. It is fairly clear that these inequalities must reduce to equalities for an optimum design. For example, if $Y > m+n-1$, then we could reduce Y or increase m or n without violating 4.3. Reducing Y or increasing m or n would reduce S according to 4.2. Hence, for the optimum designs, 4.3. and 4.4 must reduce to equalities.

We can use these equalities to eliminate Y and express S as a function of m and n. Minimization then becomes a straightforward problem in elementary calculus. We have

$$S = (m+n-1)\left[M+N+\frac{M}{m}\frac{N}{n}\right] \quad \text{in the dynamic case} \qquad 4.5$$

$$S = n\left[M+N+\frac{MN}{n^2}\right] \quad \text{in the static case} \qquad 4.6$$

Setting $\partial s/\partial m$ and $\partial s/\partial n$ equal to zero, we get for the dynamic case

$$m^2 = \left[\frac{MN}{M+N}\right] \quad (1-\frac{1}{n}) \qquad 4.7$$

$$n^2 = \left[\frac{MN}{M+N}\right] \quad (1-\frac{1}{m}) \qquad 4.8$$

and for the static case

$$m = n = \sqrt{\frac{MN}{M+N}} \qquad 4.9$$

Although 4.7 and 4.8 are difficult to solve for m and n, we observe that if m and n are much greater than unity, we may make the obvious approximations on the right-hand sides, in which case 4.7 and 4.8 reduce to 4.9. It should be noted that even 4.9 is itself an "approximation" in the sense that it usually does not yield integral values for m and n; so that the calculated values must be rounded off to the nearest integer anyway. Hence for practical purposes, the static and dynamic cases yield the same values of m and n.

Substituting 4.9 into 4.5 and 4.6, we get, for the dynamic case

$$S = 4\sqrt{MN(M+N)} - 2(M+N) \qquad 4.10$$

and for the static case

$$S = 2\sqrt{MN(M+N)} \qquad 4.11$$

Note that the second term in 4.10 is much smaller than the first; in fact, dropping this term is equivalent to using Y = m+n instead of Y = m+n-1, which introduces only a small error if m and n are large. Hence, to a first approximation, we see that the dynamic case takes about twice as many switches as the static case. In other words, knowing all connections beforehand allows a saving of a factor of two.

4.4 Divisibility Considerations

All the formulas in the previous section are approximate. For example equation 4.9 does not usually yield integer values of m and n. Even if the values turn out to be integers, they may not be divisible into M and N.

To get around this, one does the obvious things: round off the calculated m and n to the nearest integer, and then round M and N up to the next larger multiple of this value. This gives a matrix slightly larger than required, allowing a few more inputs and outputs to be terminated in the matrix. However, for matricies sufficiently large to be of interest, the percentage error in 4.10 and 4.11 is small.

4.5 Comparison with Rectangular Matrix

To see how fast the number of switches increases as a function of matrix size, we may express S in terms of N and E (see Chapter 2) rather than N and M. This gives

$$S = 4N^{3/2}\sqrt{E(1+E)} \quad \text{(dynamic case)} \qquad 4.12$$

$$S = 2N^{3/2}\sqrt{E(1+E)} \quad \text{(static case)} \qquad 4.13$$

Since $S = EN^2$ for a rectangular matrix (see Chapter 2), it follows that for small systems, a rectangular matrix is more economical, while for large systems, the three-stage matrix has the advantage. The "crossover" point may be found for the static case by setting

$$2N^{3/2}\sqrt{E(1+E)} = EN.^2$$

Solving for N, we get $N = 4\left(\frac{1+E}{E}\right)$ 4.14

For E = 1, this gives a crossover value of N = 8. For E>1, the crossover value is even smaller. Hence even fairly small rectangular matricies may be profitiably replaced by three-stage matrices. However, small matricies tend to be dominated by divisibility considerations (rounding m and n to thenearest integer and increasing M and N to the next larger multiple of this value will introduce more error if m, M, and N are small). Furthermore, the existence of fanout alters the crossover value (see Chapter 5).

4.6 Summary

The results of this chapter may be summarized as follows:

a. The number of switches required by a three-stage matrix is given approximately by $S = 2N^{3/2}\sqrt{E(1+E)}$ (for static, non-fanout programs).

b. To handle the same programs dynamically (in terms of the telephone analogy, to allow any one of the inputs to "call" any of the outputs at any time without breaking previous connections) approximately twice as many switches are required.

c. In either case, the number of switches increases in proportion to the 3/2 power of the matrix size, rather than the square.

d. The optimum values of m and n are the same for either the static or the dynamic case: $m = n = \sqrt{N}\sqrt{\frac{E}{1+E}}$

Thus the number of inputs per input block (and outputs per output block) varies as the square root of the matrix size: doubling the size of the computer should result in approximately 41% more input blocks, each having 41% more inputs.

Note, in passing, that the expansion factor of a three-stage matrix is the product of the expansion factors of the individual matricies of which it is composed. The input blocks have expansion factor $E_1 = Y/n$; the middle blocks have the expansion factor

$$E_2 = \frac{Z}{X} = \frac{M/m}{N/n}$$

and the output blocks have $E_3 = m/Y$. The product of these factors reduces to M/N, the expansion factor of the overall matrix. This fact is somewhat analogous to the fact that the gain in a multi-stage amplifier is the product of the individual stage gains. The optimization of the matrix design may be thought of as determining the most economical distribution of expansion among the three stages. In terms of expansion factors of the submatricies, we may state the following:

e. In the optimally designed static case, the input and output blocks are identical square matricies, and the middle blocks have the same expansion factor as the overall N-by-M matrix. In other words "all the expansion takes place in the middle blocks." In the dynamic case, the input blocks are approximately 1:2 expanders; the output blocks are approximately 2:1 concentrators, and the middle blocks have the same expansion factor as the overall matrix.

f. In either the static or the dynamic case, half the switches are in the middle blocks, with the other 50% divided between the output blocks and input blocks in the ratio E:1. This fact will be important later when we consider the effect various modifications to the three-stage matrix: the middle blocks are the most prolific users of switches; the output blocks are next (since $E \geq 1$); and the input blocks use the least.

5. THREE-STAGE MATRIX THEORY WITH FANOUT

The optimum design problem for three-stage matrices without fanout has been solved in Chapter 4. Most of the contents of that chapter are not new; the basic ideas were part of the published literature at the beginning of this project. However, comparatively little had been done to obtain similar results in the case of programs with fanout.

It is fairly clear that fanout increasesthe possibility of blocking; in fact, programs with fairly light fanout exist for which the conditions $Y \geq m$ and $Y \geq n$ are not sufficient to ensure that the program can be implemented (Examples will be given later).

The design parameters of a three-stage matrix, as pointed out in Chapter 4, are Y (the number of middle blocks), n (the number of inputs per input block) and m (the number of outputs per output block). The problem is to choose these parameters so as to minimize the number of switches while still allowing a significant class of programs to be implemented. Optimizing a function of three variables by trial and error is a tedious process, especially if the constraint is not specifed precisely. The previously published studies attack the problem by fixing m and n according to Equation 4.9:

$$m = n = \sqrt{\frac{MN}{M+N}}$$

This leaves only the single parameter Y to be determined. Obviously, the greater the value of Y, the more switches there are in the matrix and the more likely thematrix is to be adequate. Hence, the problem becomes one of determing the smallest number of middle blocks which will still allow a significant class of programs to be implemented. This should not be too difficult to determine by examing sample programs. This is the approach suggested by Marshall and Hagerbaumer [2] and, to an extent, by Ocker [3], but they did not actually carry out the details.

The use of equation 4.9 to determine m and n is apparently based on the assumption that if it gives optimal results for the non-fanout case then it is problably optimal (or near-optimal) even in the case offanout. In fact, equation 4.9 gives the correct answer to two different problems (static and dynamic) and hence it seems reasonable to assume that it provides "efficient" values of m and n for all cases.

This assumption turns out to be incorrect. In this chapter, the theory of Chapter 4 will be extended to cover fanout, and it will be seen that in the presence of fanout, we should choose $Y = m > n$ for an optimal design. Formulas for optimal values of m, n, and Y are developed in this chapter. The main result is that the number of switches must be increased by a factor of $\sqrt{2}$ to cover the case of fanout. The analysis is performed by constructing a "worst case" program and determining the number of middle blocks required.

5.1 The Programming Array

A program was defined in Chapter 2 as a set of connection statements, or, equivalently, as a function from outputs to inputs. For purposes of analysis it is desirable to present programs in a visual form. If the N matrix inputs and the M matrix outputs are arranged in a rectangular array (with each input corresponding to a row and each output corresponding to a column) then an X may be marked on the array for each connection to be made; that is, an X in row i, column j indicates that the i-th input is to be connected to the j-th output. Thus a program might also be defined as an array whose entries are binary digits. Note that if the program is to be implemented on a rectangular matrix of switches, the X's indicate which switches are to be closed. The restriction that no two matrix inputs connect to the same matrix output means that no column contains more than one X. Any pattern of X's satisfying this constraint represents a legitimate program.

If the program is to be implemented on a three-stage matrix, then a few modifications can be made to simplify programming array. Instead of numbering the inputs 1, 2, ... N, they should be grouped into input blocks and doubly indexed. Of course, the total number of inputs remains the same, no matter how they are indexed. As far as outputs are concerned, we can greatly reduce the size of the array by observing that we really don't need to specify which inputs are connected to which outputs, but only which inputs are connected to which output blocks. Since an output block is a complete rectangular matrix, it follows that if an input is connected (via an appropriate middle block) to any terminal on a given output block is can easily be connected to any or all of the outputs on that block. Hence, instead of one column per output, we need only one column per output block, which reduces the size of the array by a factor of m. The number of X's in a column cannot exceed m (the number of outputs per output block). It should be clear that any array of X's that meets this requirement represents a legitimate program for that matrix.

5.2 Implementing a Program

A program is implemented by choosing an appropriate path for each connection. In terms of the programming array, all that needs to be specified is which middle block is used to make which connection. Once this is determined, the path from the input blocks to the output blocks is completely defined. If themiddle blocks are numbered $1, 2, \ldots Y$, then a program may be implemented by erasing each X from the programming array and replacing it by the number of the middle block used to make the connection. The assignment of middle blocks must satisfy the following restrictions:

5.2.1 No number may appear more than once in a column. This is because each middle block has but one connection to a given output block.

5.2.2 No number may appear more than once in any input block except in the same row. This is because each row represents an input, and a given input block has but one connection to a given middle block.

Any assignment of numbers that meets these restrictions represents a valid implementation of the program.

Figure 5.1 illustrates these concepts for a small matrix with 3 input blocks of 4 inputs each, and 6 output blocks of 4 outputs each. This is a legitimate program, since no column contains more than four entries. It is also a "worst case" program in the sense that every column has exactly four entries. The reader should be able to verify that 5.1b represents a valid implementation of the program with six middle blocks. (It will be proved later that six middle blocks are really necessary for this matrix, despite the fact that m and n are only four).

If an input has to fanout to several outputs, this fanout could take place at several points in the matrix: within an input block, within a middle block, or within an output block. It is worthwhile to take a few minutes to observe what this fanout looks like on the programming array:

INPUT BLOCK	INPUT	OUTPUT BLOCK 1	2	3	4	5	6
1	1	X	X		X		
	2	X		X		X	
	3		X	X			X
	4				X	X	X
2	1	X					X
	2		X			X	
	3	X					X
	4		X			X	
3	1			X			
	2				X		
	3			X			
	4				X		

5.1a

INPUT BLOCK	INPUT	OUTPUT BLOCK 1	2	3	4	5	6
1	1	1	1		1		
	2	2		2		2	
	3		3	3			3
	4				4	4	5
2	1	3					6
	2		2			1	
	3	4					4
	4		5			5	
3	1			4			
	2				2		
	3			5			
	4				3		

5.1b

If a particular input fans out within an input block, then two or more different numbers appear in the corresponding row (e.g. the first input in the second input block in Figure 5.1b).

If a particular input fans out within a middle block, then the same number appears several times within the row (e.g. the first input in the first input block in Figure 5.1b).

If any input fans out within an output block, then the column corresponding to that output block must have less than m entries. Conversely, if a column has fewer than m entries, then the corresponding output block must either have fanout or idle outputs or both.

Incidentally, fanout within an output block is desirable, since it reduces the number of X's in a column and hence makes it more likely that the program can be implemented. This means, forexample, that a summer should not have all its inputs on one output block; they should be "spread" among several output blocks to enhance the possibility of output block fanout.

Input block fanout is occasionally necessary; in fact, it is for this reason that it is undesirable to take Y=m=n, as we did in Chapter 4. Consider, for example, the program in Figure 5.2 the matrix has N=8, n=4, M=12, m=4. Since no column has more than four entries, this is a permissible program for this matrix. However, four middle blocks are not enough This can be seen as follows:

	1	2	3
1	X	X	X
2	X	X	X
3	X	X	X
4	X		
1		X	
2			X
3			
4			

5.2a

	1	2	3
1	1	1	1
2	2	2	2
3	3	3	3
4	4		
1		?	
2			?
3			
4			

5.2b

The first column contains four entries. Hence it must use at least four middle blocks. We may assume without loss of generality that they are numbered in order 1, 2, 3, 4. Numbering them in any other order would make no difference, since it would merely amount to a re-labeling of the middle blocks. Since four middle blocks are used in the first input block, it follows that if only four middle blocks are available, then there can be no fanout within input block number 1. Hence, the remaining entries in input block number 1 are forced, that is, we must use the same number that is already present in the row.

Now look at the unassigned entries in input block number 2. Since the same number cannot appear more than once in a column, both entries in the second block must be 4. But this is impossible since they are in different rows. Hence, four middle blocks are insufficient. Note that it is the presence of heavy fanout in a single input block with all its inputs in use that causes this difficulty. Since many analog programs use most of the components available, it is quite possible that there will exist at least one input block with all its inputs in use. If Y=n, then that input block cannot have any fanout. If the inputs in this block must fanout to several output blocks, there will be a lot of forced choices of the type encountered in the above example. To avoid this difficulty, we should take $Y \geq n$, so that the input blocks are expanders, rather than square matrices.

Now how about the other condition encountered in Chapter 4, namely $Y \geq m$. Should this also be replaced by $Y > m$? At this point, we cannot tell (it will turn out later that the answer is "no", that is, we should have Y=m for an optimal design). The point to note here is that the above argument does not apply to output blocks, since fanout within an output block helps us, rather than hurting us.

In this context, it is worth noting that in the absence of fanout, there is a symmetry between inputs and outputs. In fact, a non-fanout program is a one-to-one correspondence between some of the inputs and some of the outputs. Hence if we interchange the words "input" and "output" and interchange M and N, the same formulas apply. This explains why formula 4.9 is symmetric with respect to m and n.

For programs with fanout, there is a fundamental difference between inputs and outputs, in that one input can connect to many outputs, but not vice versa. This makes it intuitively reasonable that one should expect m = n in the non-fanout case, but $m \neq n$ when fanout is allowed.

5.3 Blocking

Two inputs will be said to block each other if they must go to the same output block. In terms of the programming array, this means that they have an X in the same column. A set of inputs will be called a blocking set if any two inputs in the set either block each other or are in the same input block. The reason for the definition is given in the following theorem:

Therorem 5.1 If two inputs in a blocking set do not fanout within their input blocks, then they cannot share the same middle block.

Proof: If the two inputs are in the same input block, then they obviously cannot share the same middle block , regardless of whether they have input block fanout or not. If they are in different input blocks, then, by definition of a "blocking set," they must have connections to the same output block. If they don't fanout within their respective input blocks, then each is connected to just one middle block. But one middle block cannot connect them both to the same output block, so they must use different middle blocks.

The "worst case" of blocking occurs when a large blocking set is concentrated in a small number of input blocks. This is seen in the following theorem due to Marshall and Hagerbaumer [Reference 2]:

Theorem 5.2 Suppose a blocking set contains P inputs, and suppose these inputs are in P/n input blocks (so that each input block is "full" i.e. all its inputs are in use). Then the number of middle blocks required must satisfy the condition

$$Y \geq \frac{2}{P^{-1}+n^{-1}} = \frac{2Pn}{P+n} \qquad 5.1$$

Proof: Let N_f be the number of inputs in the set with fanout in the input block and N_o be thenumber without input block fanout. Then

$N_o \leq Y$ by Theorem 5.1

$N_f \leq (P/n)(Y-n)$ because there are Y-n unused inputs in each input block.

Hence $P = N_o + N_f \leq Y + (P/n)(Y-n)$
Solving for Y, we get condition 5.1

Note that this condition is necessary, but has not been proved sufficient. However, extensive experience indicates that it is a sufficiently strong condition so that in practice, it is usually sufficient. For example, the first eight inputs in Figure 5.1 form a blocking set concentrated in the first two input blocks. By formula 5.1, we must have $Y \geq 5.33$. Since Y must be an integer, this implies $Y \geq 6$. Since the program is actually implemented with 6 middle blocks in Figure 5.1, this number is both necessary and sufficient.

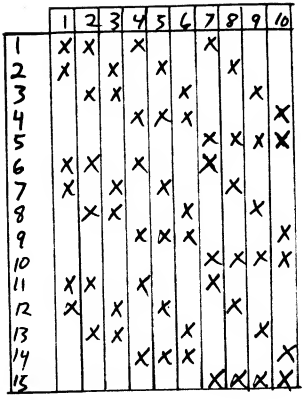
Incidentally, the program in figure 5.1 was implemented simply by proceding column by column, filling up each column with numbers, keeping the necessary restrictions in mind. The necessity for the fifth middle block becameevident in column 2 (this is the same sort of blocking illustrated in Figure 5-2). The sixth block was not needed until the very last column. Thus the program almost succeeded with five, but "almost" isn't good enough: Theorem 5.2 shows that six are really necessary.

To use this formula as a design tool, we need some means of relating P (the size of a maximal blocking set) to the matrix parameters. This relation is derived in the next section.

5.4 Construction of "Worst Case" Programs

The greater the value of P, the more middle blocks are required. Construction of a worst case program therefore requires determination of a maximal blocking set. If we enter an X in every row and every column of the programming array, then every input blocks every other input, but this is not a legitimate program since the number of X's in a column cannot exceed m. Hence we want to generate the maximum amount of blocking without too many entries in a column.

If one input blocks another, then there must be some column with at least two entries (namely the column in which blocking takes place for these two inputs). It seems reasonable to ask how many more inputs we could add without increasing the number of column entries beyond two. Each new input must block all previous entries in the set, and this blocking must not take place in one of the columns that already has blocking, since that would mean three entries in a column. Hence new columns are needed for each new input to block the preceding ones.



5.3

Figure 5.3 shows the construction. If inputs 1 and 2 block each other, they must have entries in the same column, and we might as well call that column 1. If input 3 is to block both inputs 1 and 2 without more than two entries in a column, then we must add two more columns (one for input 3 to block input 1 and one for input 3 to block input 2.). We may as well label these columns 2 and 3. Each new input needs one new column for each of the preceding ones. Adding the fourth input requires adding three more columns, and adding the fifth input requires four additional columns. Therefore, any set of five mutually blocking inputs, with only two entries per column must look like the pattern of the first five rows in Figure 5.3 (except for re-arrangement of rows and/or columns.)

The total number of columns required may be obtained by summing an arithmetic progression. Table 5.1 illustrates the trend and gives the general formula.

NUMBER OF BLOCKING INPUTS (ROWS)	NUMBER OF COLUMNS REQUIRED
2	1
3	3
4	6
5	10
6	15
U	U(U-1)/2

Table 5.1

We may continue this construction until we run out of columns. Since the number of columns is Z(=M/m), we have

$$U(U-1)/2 = Z.$$

Solving for U, we obtain, by quadratic formula:

$$U = 1/2 \left[\sqrt{8Z+1} + 1\right]$$

Of course, this formula is exact only in case Z happens to be of the form U(U-1)/2, so that the calculated value of U turns out to be an integer. However, for large matrices, Z is sufficiently large so that we do not introduce much error by rounding U to the nearest integer.

In fact, for large Z, we can make the approximation:

$$U = \sqrt{2Z}.$$

This formula gives (approximately) the maximum number of mutually blocking sets with only two entries per column. Since we are allowed m entries per column, we may repeat this pattern m/2 times (if m is even) obtaining $P = \left\lfloor \frac{m}{2} \right\rfloor \sqrt{2Z}$ mutually blocking inputs. Figure 5.3 illustrates this construction for U = 5, Z = 10 and m = 6. Since this formula was obtained by generating the maximum number of blocking inputs for a given columnlength, it appears to be the worst-case value of P or at least close to it, and experience bears this out.

Eliminating Z in favor of the more familiar parameter m, we obtain

$$P = \sqrt{Mm/2}$$

Using this value of P in Theorem 5.2, we obtain

$$Y \geq \frac{2}{n-1+\sqrt{2/Mm}}$$

as a necessary condition to prevent this type of blocking.

5.5. Construction of an Optimal Three-Stage Matrix With Fanout

A three-stage matrix should minimize the number of switches, given by

$$S = Y\left(M+N+\frac{MN}{m}\right) \qquad 5.1$$

subject to the constraints

$$Y \geq n \qquad 5.2$$

$$Y \geq m \qquad 5.3$$

$$Y \geq \frac{2}{n-1+\sqrt{\frac{2}{Mm}}} \qquad 5.4$$

The first two conditions are necessary for the same reasons given in Chapter 4, and the third was derived in the last section. Since the right-hand side of 5.4 is a monotonic increasing function of m and n, if follows that all three constraints impose upper bounds on m and n and lower bounds on Y, as expected (see chapter 4).

When a system to be optimized is subject to inequality constraints, then, at the optimum point, some of these inequalities reduce to equalities, while the others remain strict inequalities. Furthermore, those that remain strict inequalities do not affect the optimum point, since a small change in the optimum point could be made without violating them, which means that the optimum point must also be optimum with respect to a reduced set obtained by ignoring all the inequalities that turn out to be strict.

Forexample, suppose we ignore condition 5.4 and optimize the system subject to 5.2 and 5.3 only. Then if we take the optimum values of m, n, and Y and substitute into 5.4 and find that it is satisifed also, then the new optimum is the same as the old and the additional contition (5.4) has no effect on the optimum. Conversely, if 5.4 is <u>not</u> satisfied, then this means that the new condition (5.4) <u>does</u> affect the optimum, and in fact, 5.4 must reduce to a strict equality.

Now the optimum in the absence of 5.4 has already been calculated in Chapter 4. For this optimum point, Y = m =n. Hence, substitituing into 5.4, eliminating Y and n, we get

$$m \geq \frac{2}{m-1+\sqrt{\frac{2}{Mm}}}$$

which reduces to

$$\frac{M}{m} \leq 2.$$

Since M/m is the number of output blocks, it follows that for any matrix with more than two output blocks (and this includes all matrices of practical interest), this inequality is violated. In other words, if we ignore the new condition 5.4 and solve the optimization problem subject <u>only</u> to 5.2 and 5.3 then the resulting optimum point does <u>not</u> satisfy 5.4. Hence the introduction of 5.4 <u>does</u> affect the optimum point, which, in turn, implies that 5.4 must be an <u>equality</u> at the optimum point.

Further analysis (details omitted here) shows that, in fact, 5.3 also reduces to an equality at the optimum point, while 5.2 remains a strict inequality. (In fact, we have already seen that $Y \geq n$ is necessary to allow fanout within an input block).

Using the two equalitites (5.2 and 5.3) allows us to eliminate two of the three variables and reduce the problem to optimization of a function of one variable. Substituting $Y = m$ in 5.1, we get

$$S = m(M+N) + \frac{1}{n} MN \qquad 5.5$$

Solving 5.4 for $\frac{1}{n}$ (remembering that 5.4 is actually an equality at the optimum point) we get

$$\frac{1}{n} = \frac{2}{m} - \sqrt{\frac{2}{Mm}} \qquad 5.6$$

Substituting 5.6 into 5.5 yields

$$S = m(M+N) + MN\left(\frac{2}{m} - \sqrt{\frac{2}{Mm}}\right) \qquad 5.7$$

Setting $dS/dm = 0$ and simplifying,

$$(M+N)m^2 + N\sqrt{\frac{M}{2}}\sqrt{m} = 2MN \qquad 5.8$$

The left hand side of equation 5.8 is a monotonic increasing function of m for fixed values of M and N. Hence there is a unique positive root m, which can easily be obtained by iteration. Substituting this value of m into equation 5.6 allows us to solve for n, thus yielding the optimal design values.

As an example, consider the case $N = 60$, $M = 144$.
Equation 5.8 yields $m = 8.8$ and equation 5.6 yields $n = 5.3$. Thus we should have $m = 8$ or 9 and $n = 5$ or 6. All these values are, in fact, divisible into M and N, so that we do not have to increase M or N to allow the inputs and outputs to be grouped into the appropriate number of blocks.

Tabulating the four possible cases, we get the following results (Table 5.2):

m	n	X=N/n	Z=M/m	Y=m	S=Y (M+N+XZ)
8	5	12	18	8	3360
8	6	10	18	8	3072
9	5	12	16	9	3564
9	6	10	16	9	3276

Table 5.2 Possible Candidates for Optimum Design (N=60;M=144).

Examination of the table indicates m=8, n=6 yields the optimum, and requires 3,072 switches. By contrast, a 60 by 144 rectangular matrix requires 8,640 switches.

5.6 The Asymptotic Formula

For large matrices, a simple approximate solution is possible. Since the first term in equation 5.8 is proportional to m^2 and the second to $\sqrt{m}$, it follows that the first term predominates for large m. Ignoring the second term, we may solve for m, obtaining

$$m = \sqrt{\frac{2MN}{M+N}} \qquad 5.9$$

This is of the same form as formula 4.9, which was obtained in Chapter 4 for the non-fanout case, except for the additional factor in the numerator. To calculate n approximately for large matrices, we observe that equation 5.6 can also be written

$$\frac{1}{n} = \frac{2}{m}\left(1 - \sqrt{\frac{m}{2M}}\right) = \frac{2}{m}\left(1 - \sqrt{\frac{1}{2Z}}\right) \qquad 5.10$$

where Z (=M/m) is the number of output blocks. For large matricies, Z becomes large so that the first term dominates, and we have, approximately:

$$\frac{1}{n} = \frac{2}{m}$$

and hence

$$n = \frac{m}{2} = \sqrt{\frac{MN}{2(M+N)}} \qquad 5.11$$

Subsituting equation 5.9 (for m and Y) and equation 5.11 (for n) into equation 5.1, we get

$$S = 2\sqrt{2}\sqrt{MN(M+N)} = 2\sqrt{2}N^{3/2}\sqrt{E(1+E)} \qquad 5.12$$

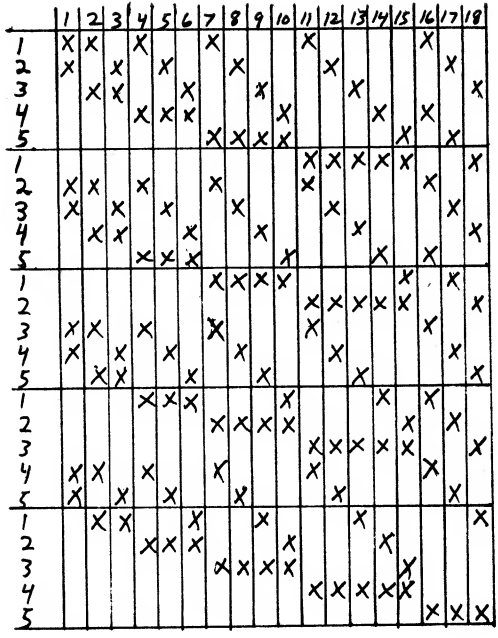
Thus a three-stage matrix optimized for handling programs with fanout takes about 41% more switches than the non-fanout case. (See Chapter 4).

5.7 Adequacy of the Three Stage Matrix

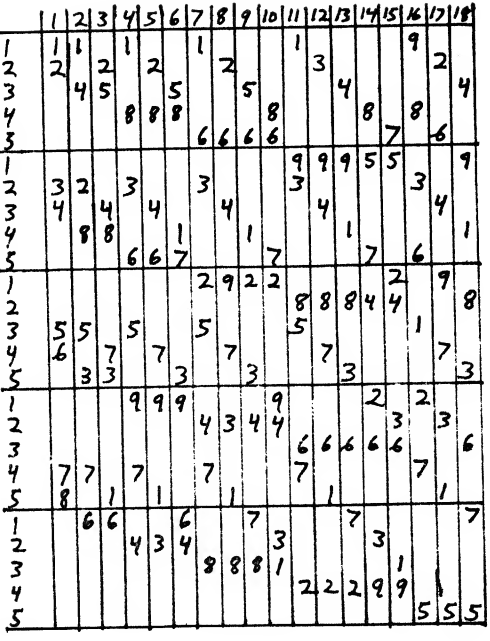
It should be remembered that the system has been optimized subject to three constraints (5.2, 5.3, and 5.4) which are known to be necessary, but have not been proved to be sufficient. However, experience indicates that a matrix designed according to these rules will actually be adequate for most practical problems.

As an illustration, consider what the "worst case" construction looks like on the matrix described above (N=60; M=144; n =5; m =8). Figure 5.3 shows the construction for the case where Z (the number of output blocks) is of the form U(U-1)/2. Such a number is called a triangular number (because a number of objects can be arrangedin a triangle if and only if the number is of this form- for example, the pins in bowling). If Z is not a triangular number, then the construction of figure 5.3 cannot be carried out exactly, and all computations based on this construction are only approximate.

In fact, since a number of approximations were made in deriving the optimal design formula, it turns out to be possible to find a program requiring nine, rather thaneight middle blocks. Figure 5.4 illustrates this program. Since 15 is a triangular number, the first 15 columns allow 6 mutually blocking inputs to be generated with only two entries per column. Repeating this pattern four times gives 24 mutually blocking inputs which fill the first 15 columns, with the maximum permissible number of entries (eight) in a given column. The remaining three columns are just enough to provide enough entries to allow the 25th input to block each of the first 21. (Note that in order to use Theorem 5.2, it is only necessary that the 25th input should block the first 20 ; it need not have any column entries in common with the inputs in the same input block, since it cannot share a middle block with them in any case).



5.4 a



5.4 b

Applying Theorem 5.2, we find P = 25 entries in 5 input blocks with 5 in a block. According to the theorem, we see that $Y \geq 8.33$ is necessary. Since Y must be an integer, it follows that at least nine middle blocks are necessary. Furthermore, nine are also sufficient, as can be seen by examining Figure 5.4b which implements the program with nine middle blocks.

Thus, because of approximations and round-off errors, the recommended matrix design is not capable of handling every possible program with Y = m = 8 middle blocks, Of course, we could design the system with nine middle blocks, which would add only and additional 420 switches to the matrix, but is this really necessary? Figure 5.4 is a legitimate program in the sense of Chapter 2, but is it really a possible program on an actual analog system? How likely is such an extreme case to occur in practice?

Notice that although the matrix has 12 input block s of 5 inputs each, only 25 of the inputs are used, and they are all concentrated in the first 5 input blocks. Note also that every matrix output is used (since every column has eight entries). How likely is a program to use every matrix output and yet use less than half of the matrix inputs? Remembering that component outputs are matrix inputs and vice versa, it follows that Figure 5.4 corresponds to an analog program in which every input on every component is connected to something, and yet over half of the component outputs are not connected to anything

Since this sort of program is not feasible on an analog computer, it seem that a more reasonable "worst case" program would be one that used every matrix output and every matrix input; even this "worst case" is worse than what we are likely to encounter in practice.

Suppose we repeat the "worst case" construction, subject to the restriction that every row in the matrix should contain at least one entry. These extra entries reduce the number of column entries that are available for generating blocking sets. To provide these necessary entries without filling up any column more than necessary we may enter them diagonally as in Figure 5.5. Two diagonals plus a few extra entries at the bottom (in the last three columns) fill up the bottom rows of the matrix and still allow the basic blocking pattern to be repeated three times in the upper rows.

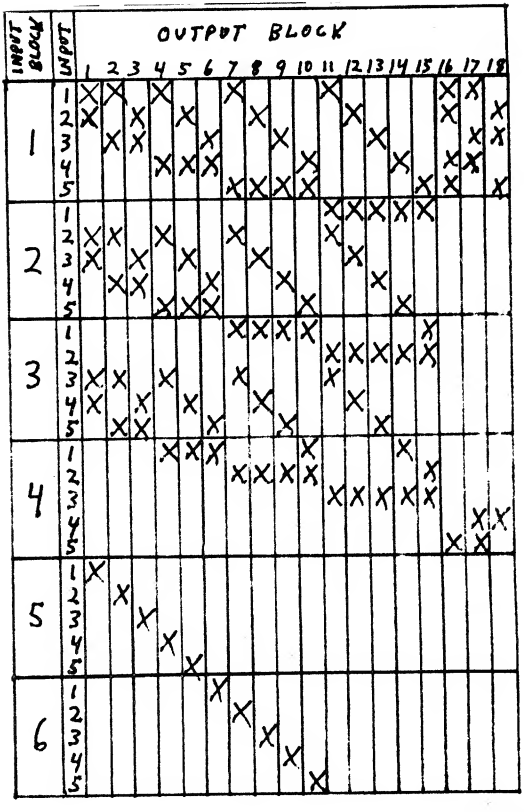


Figure 5.5a (First six input blocks)

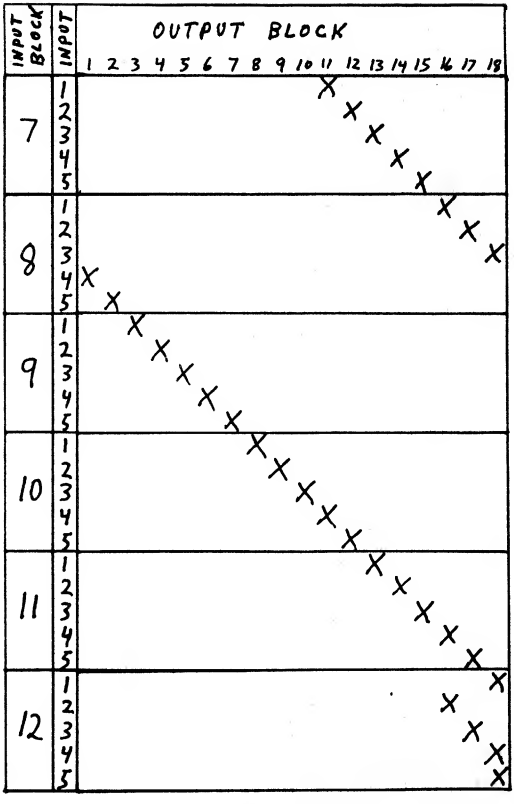


Figure 5.5a (Last six input blocks)

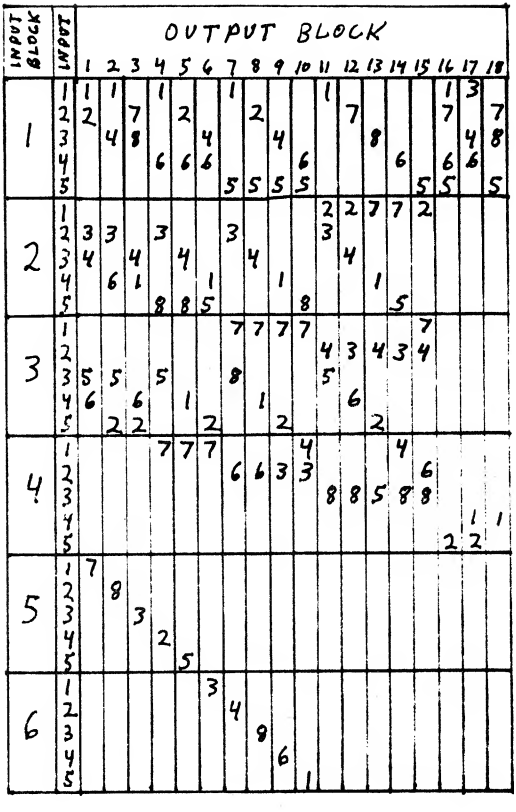


Figure 5.5 b. (First 6 input blocks)

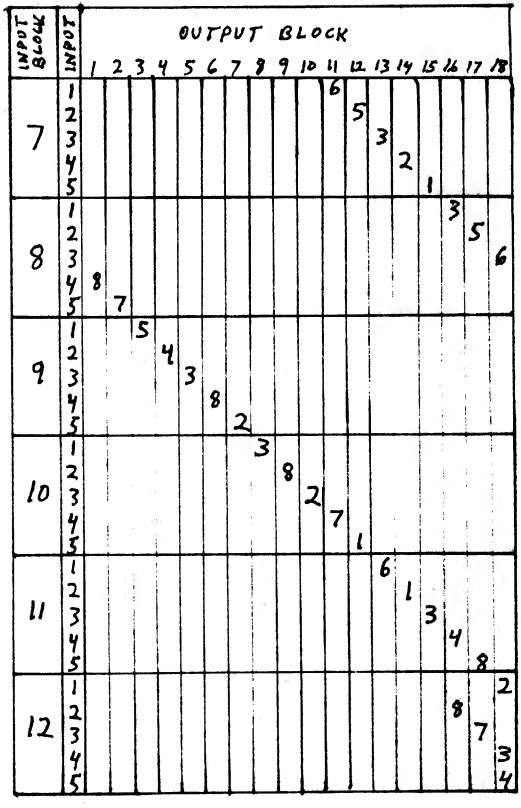


Figure 5.5 b (Last 6 input blocks)

Since the first 15 inputs form a blocking set which completely fills three input blocks, we may use theorem 5.2 with P = 15, to conclude that $Y \geq 7.5$. Since Y must be an integer, this implies $Y \geq 8$.

The first four input blocks do not form a blocking set, since the last two inputs in the fourth input block do not block any inputs in the second and third input blocks. However, even if they did, we would only have P = 20, which gives by theorem 5.2, $Y \geq 8$. Hence, in this case, theorem 5.2, is useless. It tells us only that we need at least eight middle blocks which is obvious anyway, since all columns have eight entries.

Furthermore, eight middle blocks are sufficient in this case, as shown in Figure 5.5b, Hence if we restrict the worst case construction to the worst case likely to be actually encountered in practice, (namely, all inputs and outputs in use) it appears that eight middle blocks are sufficient. Experience with actual problems submitted by NASA indicates that actual problems do not yield nearly as dense, concentrated traffic as the examples in this chapter so that eight middle blocks are more than adequate.

Incidentally, figures 5.4b and 5.5b were programmed manually, before an algorithm was developed for computer implementation. The interested reader may try covering up the solution and trying to solve 5.4 with a nine middle blocks or 5.5 with eight. Every X must be replaced by a number from 1 to 9 (for 5.4) or 1 to 8 (5.5) without using the same number twice in a given column, or using the same number twice in an input block except in the same row. The problem is not a trivial one.

5.8 Summary

The number of switches necessary for a three-stage matrix with fanout is given (approximately) by

$$S = 2\sqrt{2}N^{3/2}\sqrt{E(1+E)} \quad \text{for the static case}$$

This has the same form as the formulas for the non-fanout case derived in Chapter 4, except for the different multiplicative constant. As before, the number of switches varies as the 3/2 power of matrix size. The matrix parameters are given approximately by

$$Y = m = 2n = \sqrt{\frac{2MN}{M+N}} = \sqrt{\frac{2NE}{1+E}}$$

The matrix parameters (X, Y, Z, m, and n) all grow as the square root of the matrix size. Thus, doubling M and N (hence keeping E fixed) will increase X, Y, Z, m, and n approximately by a factor of $\sqrt{2}$.

In terms of expansion factors of the submatrices, the input blocks are approximately 1:2 expanders; the output blocks are square and hence the middle blocks have half the expansion factor of the main matrices. Compare this conclusion with comments (e) and (f) at the end of Chapter 4.

It may also be verified by direct substitution that half the switches are in themiddle blocks, just as in the non-fanout case.

5.9 Alternatives

The type of three-stage matrix studied in Chapters 4 and 5 is characterized by the fact that each middle block has exactly one connection to each input block and one connection to each output block. The theory developed in these chapters is concerned with finding the most efficient matrices of this general type. But is this type of matrix itself optimal? Perhaps it would be even more efficient to allow several connections between a given input or output block and a given middle block. Or perhaps an input block should have access to only a fraction of the input blocks and output blocks. We might say in the former case that a middle block has multiple access, and in the latter case, fractional access. The matrices covered so far in this report have had middle blocks with single access. It is worth investigation to see if this type of access is really optimal.

Consider, for example, a middle block with two connections to each input block and two connections, to each output block. Such a middle block would be more flexible than the conventional single-access middle block. In fact, such a middle block could be used twice in any column of the programming array and could be used in two different rows in the same input block. Hence it would be equivalent, for programming purposes, to two conventional middle blocks. However, since it would have twice as many inputs and outputs, it would take four times as many switches as a conventional middle block, or twice as many switches as a pair of conventional middle blocks. Hence single access is more efficient than multiple access.

How about fractional access? For concretness, assume that the inputs to the matrix are partitioned into two subsets A and B, each containing half the inputs. Similarly, assume the outputs are partitioned into equal sets C and D. Let a given middle block have access to inputs in subset A or B, but not both, and similarly for output blocks. In such a matrix the middle blocks have only half as many inputs and outputs as in the conventional design, and hence only one-fourth as many switches. However, they are less flexible than conventional single-access middle blocks so that more of them will be needed.

Implementing a given program on such a matrix can be broken down into four subprograms. To make the desired connections, we must make the conncections between inputs in subset A and outputs in subset C, and similarly, connections from A to D, B to C, and B to D. These four subprograms are independent of each other; in fact, we can really regard the matrix as partitioned into four small matrices, each of which has N/2 inputs and M/2 outputs (and hence the same expansion factor as the original). Since the size of a three-stage matrix varies as the 3/2 power of the matrix size, the four small matrices will require a greater total number of switches than a single large matrix with conventional single-access middle blocks. This reasoning can be extended to other partitionings (e.g., into unequal subsets). Thus we see that the single access structure does provide the optimal design for a three-stage matrix.

One other possibility should be considered; the possibility of using more than three stages of switching between inputs and outputs. The optimal number of stages is, in fact, a function of matrix size; the larger the matrix, the more stages should be used. The larger a three-stage matrix grows, the larger its submatrices grow. If the matrix is sufficiently large, its input blocks, output blocks, and middle blocks will be large enough to justify replacing them by 3-stage matrices.

Examination of the formulas developed in this chapter (and the specific examples given in Chapters 6 and 7) indicate that the middle blocks are the largest of the three types of submatrices. Hence they should be the first to be replaced by three-stage matrices. The crossover value at which the rectangular matrix and the three-stage matrix are equal may be found by setting

$$2\sqrt{2}N^{3/2}\sqrt{E(1+E)} = EN^2$$

Solving for N, we get $N = 8\left(\frac{1+E}{E}\right)$ This is the same formula that we saw in Chapter 4, except for the factor of $\sqrt{2}$ (due to fanout) which doubles the crossover value. The middle blocks of the recommended matrices in Chapters 6 and 7 are approximately square (E=1) so that the crossover occurs at N=16. Hence a square matrix should be made three-stage only if it is larger than about 16 by 16. The largest rectangular matrices used in Chapter 7 are 15 by 16 matrices; thus it appears that these submatrices should remain one-stage. Actually the formulas on which this crossover value is based are only approximations, valid for large matrices. Any matrix as small as 15 by 16 should be examined on its individual merits. When we do this, we see that a 15 by 16 rectangular matrix requires 240 switches. In contrast, dividing the inputs into 5 input blocks of 3 inputs each and the outputs into 4 output blocks of 4 outputs each, we get a total of 204 switches, assuming four middle blocks are adequate. Furthermore, analysis of the programming array indicates that even the "worst case" program constructed along the lines of Figure 5.3 requires only four middle blocks. Thus we could probably replace these 15 by 16 middle blocks with three stage matrices, at a saving of 36 switches per middle block.

Since the large system (two 680's) takes 12 middle blocks, the saving would be 432 switches (out of a total of about 20,000). This saving is achieved at a cost of making the overall main matrix five-stage (one stage of input block switching, three stages in the middle blocks and one in the output blocks). Since an analog signal undergoes some degradation for each stage of switching it passes through, the 2% overall switch saving probably does not justify the added complication of five stages.

6 DESIGN OF THE ANALOG CONFIGURATION (GENERAL CONSIDERATIONS)

The proposed configuration of analog hardware and switches is described in detail in Chapter 7. The present chapter is intended to explain to some extent how this proposed system evolved. There are at least three reasons for explaining the "why" as well as the "what" of the proposed system:

a. Understanding the System

The system description in Chapter 7 will be easier to follow if some of the reasons for it are understood beforehand. From this point of view, this chapter may be thought of as an introduction to Chapter 7.

b. "Proof" of Adequacy and Optimality

Is the proposed system adequate to handle most problems, and is it optimal or near optimal in terms of the number of switches? The first question is answered in Chapters 8 and 9. The second question cannot be answered with mathematical precision; I know of no way of proving that no system with fewer switches would do the job. However, the least that can be done is to document some of the "blind alleys" that were explored in order to answer some of the anticipated questions of the type, "Why didn't you do such and such?"

Actually, I am reasonably sure that the proposed system is not optimal; it would be extraordinary if there were no room for improvement. Further experience with this system will doubtless suggest further refinements and improvements. However, the design goals have been met, and further improvement should only be marginal. A 10% or 20% reduction in cost is important in designing a production system, but probably does not make the difference between a feasible and an unfeasible system.

c. Further Research

Anyone attempting to improve the system may benefit from some of the concepts and conclusions in this chapter.

6.1 The Cost Per Input or Output

In weighing one analog configuration against another, the cost of any analog component should be adjusted to account for the number

of switches it adds to the switching matrix. In other words, the cost should include a portion of the switches in the matrix as well as the cost of the component itself. This "switch cost" is dependent only on the number of inputs and outputs and not on the mathematical operation performed. From the point of view of the switching matrix, a two-input summer, a two-input integrator and a multiplier are similar; they are all two-input-one-output devices.

The exact number of inputs and outputs in a 680 depends on how it is configured (which components are "buried" and which are terminated within the matrix, how many inputs a summer has, and so on). However, the approximate number of outputs must be somewhere around 150, exclusive of pots, and the approximate number of inputs about twice that. In fact, if we add up 36 summers, 30 integrators, 39 multipliers, 18 VDFG's and 6 resolvers, we get a total of 282 inputs and 147 outputs. This total assumes an average of 3 inputs per summer and two per integrator, and assumes the resolver has three inputs (X, Y, θ) and four outputs (U, V, sin θ, cos θ). The inclusion of comparators and readout devices, which have inputs but no analog outputs will increase the input count additionally.

The actual proposed configuration has 150 outputs and 372 inputs (see Chapter 7). These values will be used in subsequent calculations. The main point to note is that even with a slightly different configuration, the number of inputs and outputs (exclusive of pots) would be approximately the same.

Now assume that the design goal (10,000 switches) can, in fact, be met. If this is the case, then we have 10,000 switches in the system and 522 inputs and outputs (150 + 372). The average number of switches per input or output works out to 19. Thus, a one-input-one-output device bears an overhead burden of about 38 switches; a two-input-one-output device bears a burden of 57 switches, and so forth.

Of even more interest is the incremental cost (i.e., the number of switches that must be added to the matrix to accommodate an additional input or output). If a rectangular matrix is used, then the incremental cost is twice the average cost, since the number of switches is proportional to the square of the matrix size. That is, if $S = EN^2$, then the average number of switches per input is EN, (which is obtained by division), while the incremental number of switches per input (obtained by differentiation) is 2EN. Actually, it turns out to be more economical to use a three-stage matrix, in which case the incremental cost is only 1.5 times the average

cost, or about 29 switches per input or output. This is only a rough estimate; the "proper" way to proceed is to use the formulas $S = MN$ (for a rectangular matrix) and $S = 2\sqrt{2}\sqrt{MN(M+N)}$ (for a three-stage matrix), and calculate $\partial S/\partial M$ and $\partial S/\partial N$. It turns out, in fact, that $\partial S/\partial N > \partial S/\partial M$, so that an additional matrix input (component output) is more expensive than an additional matrix output (component input). However, the estimate of 29 switches is the right "ballpark" in either case.

Of course this figure does not really mean that we could add a single input or an output to a three-stage matrix by adding 29 switches. Adding one extra input would mean that the new value of N would no longer be divisible by n. However, the figure gives a rough guide to the incremental cost of adding a moderate number of inputs or outputs provided the expanded matrix satisfies the appropriate divisibility conditions.

The incremental figure of 29 switches per input or output is the important one to keep in mind when deciding whether certain inputs and outputs should be buried or "terminated" in the main matrix. A detailed evaluation of such tradeoffs requires knowledge of the dollar cost of the switches (including, of course, driving logic, labor, testing, and so forth), versus the dollar value of the buried analog hardware. At present the cost of the switches is not known, but a few conclusions may be drawn without such knowledge.

For example, the recommended design commits a pot to each integrator input and each summer input. If one of these pots is terminated as a free component, then one input and one output are added. The incremental cost of this operation is an additional 58 switches. The incremental benefit is that the pot is now free to be used with other components, so that we need fewer pots overall. The proposed design uses about two times as many pots as the standard 680 because of the way they are committed. If we terminate the pots as free components, then we can reduce the pot count back down to the regular 680 complement. In the proposed design, two amplifier inputs require two pots; if the pots are terminated as free components, these two inputs will require only one pot, since some inputs may be used without pots and some may be idle in a given problem. This free pot adds one input and one output to the system, and hence 58 switches. Thus we have a tradeoff of 58 switches added versus one pot saved. If a pot is cheaper than 58 switches, then we are justified in burying the pots.

It seems likely that a pot is considerably cheaper than 58 switches. In fact, we can get an adjustable gain from 0.0000 to about 6.4 on any amplifier by using 16 switches and weighted input resistors. These resistors do not have to be precision resistors; if we set them in a closed-loop fashion, then 1/2% resistors are perfectly adequate, so that the switches themselves are the main part of the cost. This is, after all, the way one sets servo pots; not by carefully controlling the pot resistance, but by closed-loop nulling.

There are other ways to implement an adjustable gain on an amplifier. Solid-state switches may be cheaper than mechanical switches for this application, or the conventional servo-set pot may be cheaper still. If this is the case, then a pot is cheaper than 16 switches; in any case, it can't be much more expensive.

There is, incidentally, one other reason for committing pots to amplifier inputs, namely, impedance. If all outputs are low-impedance amplifier outputs, the crosstalk and contact resistance problems are lessened.

6.2 Analog Flexibility

Any automatic patching system will have to restrict the flexibility of the analog components to some extent. As a minimum, the analog programmer will have to give up his summing junctions, ungrounded pots, and various configurations of loose resistors, diodes, and capacitors hanging from the patchpanel. At this prospect, many people will say "good riddance!" Of course, the programmer should still be allowed to do what he wants to do, but he will be restricted in how he does it. He will have to express what he wants to do mathematically, rather than saying, "I'll just stick a diode in here".

Actually, there has been a trend in this direction for some time, even without the impetus of automatic patching. The introduction of patched logic replaces the need for many of the old diode and amplifier circuits that only a few years ago cluttered up the literature of the analog field. Most of the rest of the "loose diode" applications can be handled with the hard-zero limit feature of the 680. Dead space, backlash/hysteresis, absolute values, peak sampling, and similar nonlinearities can be easily and accurately programmed by means of this feature (see the 680 Reference Handbook). Of course, this circuit is not new; it's one of the oldest tricks in the book, but to my knowledge, the 680 is the first machine

to include it as a standard feature. Like logic, this circuit belongs within the machine, not hanging off the patchpanel; this is desirable in any case, and it becomes mandatory when there is no patchpanel. Like the standard 680, the proposed system includes 12 summers with this feature: Configuring a summer for this feature requires four form-A contacts.

Another obvious way to reduce the number of inputs and outputs in the matrix is to commit input inverters and output amplifiers to nonlinear components. With the present integrated-circuit amplifiers, this is desirable even on machines with patchpanels. The "waste" caused by redundant amplifiers is clearly tolerable.

Certain other schemes for giving up analog flexibility can backfire. For example, consider the following propositions:

a. It is not necessary to have multi-input integrators. In fact, many programmers prefer to generate the derivative explicitly anyway. If all integrators have only one input instead of five or six, then the size of the switching matrix is reduced.

b. "We can cut down on (or eliminate)configuration switching by not allowing integrators to be used as summers, or multipliers as dividers." This is in line with the 'black box' approach; a single-purpose component is simpler than a multi-purpose component.

c. One of the principal difficulties encountered in this problem is the high traffic density (see Chapter 3). Since blocking is less likely in a low-density program, it may be desirable to restrict the density of a program. If, for example, not more than half the components are used at any one time, then the traffic density is at most 50%. If a matrix designed for 50% density uses significantly fewer switches then one designed for near 100% density, then perhaps we should design the matrix for low-density traffic and tell the programmer not to use more than 50% of the components in any problem. Of course, this wastes analog equipment, but perhaps this is one of these cases where sacrifice of analog equipment is justified by the switch saving.

All three of these propositions have a certain surface plausibility, but, in fact, they are at best misleading and at worst simply false.

It is probably easiest to see the difficulty with Proposition (a). Suppose three inputs must be summed and integrated. If the derivative is not explicitly required, we may sum into a three-input integrator. This integrator has three inputs and one output. If only single-input integrators are available, we must use a summer (three inputs, one output) followed by an integrator (one input, one output). Thus four inputs and two outputs are tied up, rather than three and one.

Thus combination of several functions (e.g., summation and integration) is desirable, but for slightly different reasons (to save inputs and outputs, rather than analog hardware). In an ordinary analog computer, functions are combined only when it seems like a natural thing to do according to the electrical design of the unit. For example, the padded servo function generator can multiply at negligible added expense; all one needs to do is feed the padded servo cup with a problem variable instead of reference voltage. Thus one obtains a function multiplier (multiplying one input by a function of the other) at essentially the same cost as the function generator alone. Since modern computers use all-electronic function generators and multipliers, there is no such natural advantage. Hence the tendency is to terminate DFG's and multipliers as separate components.

The above reasoning also applies to Proposition (b). If I need more summers, and have only integrators left with a module, I must add another module to solve the problem and this module ties up additional switches. (The modular nature of the proposed system will be described later in this chapter and in Chapter 7. At present, it is enough to point out that the 680 is divided into six nearly identical modules, and the unused modules are available for other problems.)

Looked at another way, each component bears a "burden" representing its share of the switches in the matrix. It has access to other components through the matrix, and if it is not used, then not only is its analog hardware (amplifier, capacitors, mode control switches) wasted, but its access to other components is also wasted.

In fact, in the extreme case where the cost of analog components is negligible in comparison with the cost of the switching matrix,

it follows that every component should be capable of performing every mathematical operation consistent with the number of inputs and outputs it has. Every two-input integrator should be capable of operating as a summer, or a multiplier.

Of course such a component would contain a lot of analog hardware that would be unused much of the time, but that would be justified if the analog hardware were sufficiently cheap. In fact, the analog hardware isn't all that cheap, so this much combination of functions is not practical at present.

Note the similarity between this analysis and the analysis of proposition (a). Conventional analogs allow a component to be used for many different things only when the combination is "electrically natural" (e.g., a multiplier and a divider use the same networks, so it makes sense to convert one to the other). Since analog hardware is expensive, the proposed system allows only the conventional changes of function (integrator/summer, multiplier/divider, and so forth). Some, but not all of the multipliers have dual X^2 capability. The reason for not giving all multipliers this capability is not that it takes a lot of configuration switches (although that is true) but, more importantly, it makes the multiplier a two-input-two-output device, which enlarges the overall switching matrix unnecessarily. Hence only 9 of the 39 multipliers are allowed dual X^2 capability. My own rule of thumb, based on experience, but not on a formal survey, is that over 50% of the multiplier applications call for multiplication, less than 25% call for division, and 25% or less call for squares and square roots. The proposed system allows 30 products or quotients and 18 square or square root functions; this should be adequate for almost all applications, and examination of the NASA problems bears this out.

In summary, a component should be allowed to be re-configured for different functions by internal "configuration switches" whenever this is "electrically natural" (i.e., uses essentially the same analog networks for the various configurations.)

Note, incidentally, that whenever a re-configuration of a component is not electrically natural, then we really have two separate components sharing the same input and output terminals. For example, a combination integrator/multiplier would consist of two essentially separate components. The only analog hardware they would share

would be the output amplifier, which would be a small portion of the total cost. The real reason for associating such components together is to allow them to share input and output terminals and hence share access to the other components. If they share this access, then they cannot both be used in the same problem; at any time at least one of them must be idle. Thus a "combination integrator/multiplier" is really just a way of allowing an idle integrator to "give" its switches (i.e., its access to other components) to another component (in this case, a multiplier). Alternatively, this component could also be regarded as a multiplier, which, if unused, can "give" its access to an integrator.

Of course, the switches committed to an idle component are wasted, and it is certainly desirable to minimize this waste. This goal is achieved by the use of a three-stage matrix. Not only does it use fewer switches than the rectangular matrix, but it wastes fewer of them if any component is idle. In a rectangular matrix, every switch is committed to two components: it connects an input of one component with an output of another. Hence it is idle if either component is idle. In the three-stage matrix, half of the switches are in the middle blocks (see Chapters 4 and 5) and hence are not committed to any specific component. Furthermore, the other switches (those in the input and output blocks) are each committed to only one component, so that the probability of a given switch being idle is much less.

For example, consider the matrix discussed in Chapter 5 (60 matrix inputs and 144 matrix outputs). If a rectangular matrix were used, each of its switches would be committed to a specific input and output. Suppose a component with two inputs and one output is idle. The two component inputs are connected to two of the 144 matrix outputs, and each of these has 60 switches associated with it (two columns of the 60 by 144 rectangular matrix or 120 switches in all). The component output is one of the matrix inputs, and has 144 switches associated with it. This gives a total of 120 + 144 - 2 = 262 switches (the two switches which are common to the row and two columns are subtracted in this computation so that they don't get counted twice).

In contrast, in the three-stage matrix configuration, the component output terminates in an input block, and has 8 switches associated with it (to connect it to each of the eight middle blocks).

The two component inputs terminate in 8 by 8 output blocks, where each has 8 switches associated with it. Hence if this component is idle, only 24 (rather than 262) switches are rendered useless. Thus, the three-stage matrix provides "shared access" capability without constructing such unnatural "components" as integrator/multipliers.

Consideration of idle components leads to an investigation of Proposition (c). If we design a matrix for 50% traffic, then only half the components can be used at one time, and a given problem will require a computer twice as large. Since the number of switches grows faster than linearly with matrix size, it appears extremely dubious that a double-sized computer will really be more economical of switches. Suppose a problem requires 20 integrators and 30 multipliers. Following the "50% traffic density" suggestion, we put this problem on a machine with 40 integrators and 60 multipliers. If we succeed in programming the problem on this machine, then half the analog components are idle. If we remove these components and the switches committed to them, we obtain a reduced system with only 20 integrators and 30 multipliers. It should be clear that the problem is really being programmed on this reduced system. In other words, any system capable of handling this problem must contain at least 20 integrators, 30 multipliers, and enough switches to allow them to be connected in the appropriate manner. If we further burden the system with additional analog equipment that we know in advance we are not going to use, then the extra equipment does us no good.

This point may be stated another way: if the cost of a computer is measured solely by the cost of its computing components, there is a strong motivation to put a problem on the smallest possible computer, since larger computers are proportionally more expensive. If the switch cost becomes a significant percentage of the total, this motivation is not reduced, but rather increased, since the cost of the computer grows faster than linearly with size. Thus the switching system should be designed for heavy traffic; we should count on most of the available components actually being in use at a given time.

Of course, 100% utilization of all components is unlikely for several reasons: grouping, balance, and burying. Grouping refers to the fact that analog components come in groups, and to use any of the components, one usually ties up the whole group. In a con-

ventional analog system, the "group" is usually an entire console with its patchpanel. All of us have had the occasional frustrating experience of slaving two analog consoles together merely to use a handful of components on the second console. This gives a system with low percentage utilization due to grouping. The proposed system has smaller groups (the group is one module, which is one-sixth of the 680). Hence the waste due to grouping will be less with the proposed system than with a conventional patched analog, and the traffic density correspondingly greater. Actually, there is no absolute reason why two users couldn't share some components within the same module, but the programming becomes more awkward. It is probably best to regard an entire module as "belonging" to one programmer at a time, and not try to split it up.

The main reason for some components being idle is balance. The component requirement for a given problem is actually a multi-dimensional vector (one must specify the requirement for integrators, multipliers, summers, DFG's, and so forth). It is extremely unlikely that a given problem will use these components in exactly the same ratios as they are present on the computer. A problem with very few nonlinearities (e.g., a control problem) will need enough modules to provide the necessary summers and integrators, and many of the DFG's and multipliers in these modules will be idle. Conversely, a six-degree-of-freedom simulation needs lots of multipliers and DFG's; in such a simulation there will probably be idle integrators. The automatic patching system will have little or no effect on this cause of idleness.

A component in a conventional analog may also be idle if it is buried, that is, not terminated on the patchpanel. For example, some of the components within an electronic resolver are not fully available in the free component mode. In the proposed system there are lots of buried pots and inverters, but these are not considered "components" at all. This report considers a summer with built-in pots as a single component and the same is true of a multiplier with built-in inverters.

To summarize this section, an automatic patching system will have to "waste" some analog hardware. This is necessary, but not sufficient. As the analysis of Propositions (a), (b), and (c) shows, it is possible, unless one is very careful, to pay the price in analog hardware without gaining the expected benefit in switch saving.

The only way to waste analog components and actually enjoy a switch reduction appears to be a straightforward "burying" of components;

the "waste" is reflected in the fact that these components are provided in excess of what would otherwise be required (e.g., pots and inverters).

6.3 Modular Design

An important consideration that has not been discussed so far is the modular nature of most analog problems. Most systems can be broken down into subsystems, with many complex interconnections within a subsystem, and relatively few signals trunked between subsystems. Any good systems analyst will try to subdivide a system in this manner merely because it facilitates his analysis and understanding of the system, regardless of the type of computer he intends to use (if any).

Analog computers are also laid out in a modular arrangement, to facilitate patching and addressing, and it seems reasonable that this approach should also be used in reducing the number of switches in an automatic patching system. A modular system, as used in this report will mean a system of analog components satisfying the following conditions:

a. The system is divided into a number of subsystems called modules.

b. Considerable flexibility of interconnection is allowed between components within the same module, by means of a matrix called an internal matrix. There is an internal matrix for each module.

c. A limited number of connections are allowed between components in different modules through an external matrix.

6.3.1 Elementary Considerations

Letting S_E be the number of switches in the external matrix, S_I the number of switches in all the internal matrices, and $S_T (= S_E + S_I)$ be the total number of switches in the system, we have the following obvious comments:

a. Increasing the number of modules (for a machine of a fixed size) decreases the switch requirement in a given internal matrix at a faster than linear rate. Hence, if the number of modules is large, S_I is small, because

the internal matrices are very small (even though there are many of them).

b. On the other hand, increasing the number of modules reduces the size of each module, and thus increases the percentage of connections that must be made through the external matrix. Hence S_E is increased.

c. An optimum design must strike a balance between the above considerations. As a rough general guide, it should be clear that S_I and S_E should be approximately equal for an optimum design. That is, if S_I is considerably greater than S_E, then an increase in the number of modules would cut down S_I substantially; and this would probably more than offset the increase in S_E. Similar considerations would hold in case S_E were much greater than S_I. It is not necessary that S_I and S_E be exactly equal, but neither should be negligible in comparison with the other.

d. An important parameter in the design is the percentage of connections that can be made internally. If this percentage is less than about 50%, then the modular idea is probably not worth bothering with. Unless the majority of connections can be made internally, we are probably better off with only the external matrix.

For example, suppose exactly half the connections are made internally. The external matrix needs to have only half as many inputs and outputs as would otherwise be required. (Otherwise here refers to the case where modularization is not used, so that the "external" matrix is the only one.) Now reducing the inputs and outputs by a factor of two would reduce the external matrix by a factor of 4 if the external matrix were a rectangular matrix. However, we have seen that the external matrix is so big that the optimal design is three-stage. This means that halving the number of inputs and outputs reduces the external matrix only by a factor of $2\sqrt{2}$. Thus the external matrix has about 35% as many switches as it would need if it had to carry the burden alone. Assuming S_I is approximately equal to S_E, the total number of switches is about 70% of what would be required without modularization. Now a saving of 30% is substantial, but it is probably not enough to make the difference between feasibility and unfeasibility. Clearly if the percentage of connections made by the external matrix is much greater than 50%.

the saving is not large enough to justify the programming complexity and loss of flexibility.

Fortunately, it turns out to be possible to reduce the number of inputs and outputs on the main matrix to about 25% to 35% in typical problems.

6.3.2 Additional Questions

This section consists of "documentation of dead ends," that is, listing and discussing briefly some of the approaches that didn't work. One of these "dead ends" (three-level modularization) may not actually be completely "dead." Further experience may indicate that is is a good idea for very large systems.

a. Similar or Dissimilar Modules? Should each module contain integrators, summers, multipliers, and so forth, in the same ratio as the whole computer, or should some contain lots of multipliers and few integrators while others have an excess of integrators and few multipliers? At first, it appeared that several different types of modules were desirable, because systems rarely break down nicely into identical subsystems. For example, a nonlinear reactor may be controlled by a complex, but nearly linear, controller. If a typical problem is modularized in a natural way, it turns out that some modules require more of one type of equipment than others. However, after examining several "unbalanced" module configurations, I feel that the balanced approach is better after all. Every attempt to design a system with dissimilar modules seems to lead to a "custom design," tailored for the requirements of a particular problem, or small class of problems. Of course, it is no trick at all to design a system to solve any one problem with no switches at all--the ridiculous extreme in switch economy at the expense of flexibility. Although the distribution of different types of components appears to be a random variable, the balanced distribution is the most probable. (Standardized design, parts stocking, and production are also easier if modules are identical, or nearly identical).

b. Universal Access or "Adjacent" Access? Should each module have access, through the external matrix, to each other module, or should it have access only to a few "adjacent" modules? The external matrix can be made smaller if each module is allowed to have access to only a few of the other modules instead of all of them.

The simplest "adjacent access" system is one in which the modules are arranged in a linear array. Suppose the modules are numbered 1, 2, 3 . . K, and suppose module i has access only to modules $i + 1$ and $i - 1$. That is, module 2 has access only to modules 1 and 3; module 3 has access to modules 2 and 4, and so forth. Modules 1 and K are considered to be "adjacent," so that every module has access to exactly two modules. The convention of regarding the first and last modules as adjacent not only makes the system more symmetric, but it allows all modules to be connected in a single closed loop.

An obvious two-dimensional generalization of the procedure is to consider the modules arranged in a rectangular array. Each module would have access to four others (the ones directly above, directly below, and to the left and the right). This rectangular arrangement is strongly suggested by the appearance of a conventional patchpanel. Of course, the patchpanel is two-dimensional for obvious, physical reasons which have little or nothing to do with structure of actual problems.

If the two-dimensional approach is used, it is probably desirable to avoid treating the edges and corners of the rectangle as exceptions. This can be done by considering a module at the extreme right of the array as "adjacent" to the corresponding module on the left, and similarly for the modules on the top and bottom. This assures that each module has access to exactly four others. The method may easily be extended to three or more dimensions.

In evaluating such a structure, the obvious question is whether or not typical problems can be broken down in such a way that each subsystem "feeds" information to only a few of the other subsystems. There are certain types of problems where this limited access between modules works fairly well. For example, a design which allows the modules to be arranged in a single closed loop should be adequate for simple control problems in which there is only one major feedback loop, and all minor loops can be programmed within a module, or at most two adjacent modules. However, as the number of minor feedback loops (and even feedforward paths) grows, there is an increasing demand for modules to access other modules "remote" from them (a path between two modules that are not adjacent is analogous to a long patchcord). Even the two dimensional array is not much better in this respect.

If access is necessary between remote modules, one must link from module to module, passing the signal through many switches

and tying up access lines in each module through which the signal passes. Unless such connections are very rare (which they aren't), they tie up too many switches.

Aerospace problems are even worse in this respect than process control problems. After all, many industrial processes are linear or nearly so. (The word "linear" here does not refer to the differential equations, but to the topological interconnection of the subsystems. An assembly line is linear in this sense, and most plants are organized, at least conceptually, in such a manner that the raw materials enter at one point, are processed successively by various subsystems, and emerge at some point as a finished product. Thus there is an overall flow from input to output in an approximately linear manner, and the concept of "adjacent" subsystems has some meaning). In contrast, the various subsystems in a typical aerospace problem might consist of "integration of the translational equations," "integration of the rotational equations," "generation of the aerodynamic co-efficients," and "simulation of the on-board controller." In such problems, almost every subsystem needs to connect to many other subsystems, and the "limited access" design is not applicable. Even process control problems need a number of "long patchcords" for large feedback loops.

For obvious reasons, the concept of "access to adjacent modules only" should work nicely for partial differential equations, but it does not appear to be adequate for other applications.

Fortunately, it does not add very many additional switches to allow universal access to all modules, provided the external matrix is multi-stage, rather than single-stage. We have already seen (in the analysis of Proposition b, Section 6.2) that only a small portion of the switches in a three-stage matrix are actually committed to a specific component. If a component in one module accesses a component in another module only rarely, then the switches that provide this access are still useful for other purposes in problems that do not require access between these two modules. Thus allowing any module to access any other module is not particularly expensive. (This would not be true if the main matrix were rectangular.)

c. Multi-level Modularization?

The possibility of more than two levels of modularization should be considered. Perhaps the modules should be further divided

into submodules, with a very small internal matrix to interconnect the components within a submodule, an intermediate-size matrix to interconnect components within a module but in different submodules, and a large external matrix to make the connections between modules.

Intuitively, it appears plausible that this should become desirable at some point if the system becomes large enough. For large systems, the external matrix becomes very large, since it grows in proportion to the 3/2 power of the size of the system. If the system is very large, it is imperative to hold down the size of the external matrix by making the modules bigger. (This allows a greater percentage of the connections to be internal, and hence allows a smaller external matrix.) Thus as a system grows in size, its modules should also grow. At some point, the modules become sufficiently large that it becomes practical to divide them into submodules.

Note the similarity between this discussion and the case of the three-stage matrix discussed in Chapter 5. As the three-stage matrix grows, its submatrices grow, and at some point they become large enough to be profitably replaced by three-stage matrices also.

I do not think that the systems proposed in this report are large enough to justify further subdivision, but I am not sure. The question should be left open for now (especially for the case of the two-console system) and should be re-examined later when more experience has been gained with the modular approach.

In fact, the proposed system could be considered a three-stage modularization, if one regards the configuration switches as an "internal matrix" connecting "components" (resistors, amplifiers, diodes) within a very small "submodule" (e.g., a multiplier/divider).

6.4 Determining Module Size

It would be desirable to have an analytical formula for determining the best size for a module, similar to the formulas developed in Chapter 5 for determining the best size for the input and output blocks. In the absence of such an analytical formula, the best approach is to examine analog programs to see if they have a "natural" module size. The module size is probably fairly close

to optimal if it satisfies the following criteria:

a. The total number of switches is significantly less than what would be required for a single large matrix connecting the same components.

b. More than half the connections can be made internally.

c. The total number of switches in all internal matrices (S_I) is of the same order of magnitude as the number of switches in the external matrix (S_E).

The proposed module size satisfies all these criteria. It was developed by examining a number of problems to determine the smallest module size that would allow at least half the connections to be made internally. Further experience in programming problems on the proposed system indicates that less than half the component inputs and one-third of the component outputs need to have access to the external matrix.

The module size can also be estimated roughly by the following argument. Once the decision is made to use identical or nearly-identical modules (See Section 6.3.2), we can describe the size of a module by a single component. Suppose we ask ourselves how many integrators there should be in a module. Once that question is answered, the number of summers, multipliers, DFG's, and so forth, is determined by the ratio of these components to the integrators.

As a bare minimum, a module should contain at least two integrators, so that a second-order loop can be programmed with internal connections only. A two-integrator module, however, is probably too small. Since second-order loops often occur in groups with considerable coupling between the loops within a group, it appears desirable to use a module large enough to contain at least two such loops, and possibly three. Thus from four to six integrators per module seems like a reasonable complement.

The proposed design contains five integrators per module, which allows two second-order systems and one first-order system to be internally interconnected. The choice of five integrators instead of four or six was dictated largely by the existing 680 structure. Since, by tradition, a 30-integrator machine normally has

three resolvers, and since the 680 resolver expansion rack is designed on the assumption of three resolvers per 680 console, it was desirable to have the number of modules divisible by three. Thus, the proposed module (one-sixth of the machine) contains 5 integrators, 6 summers, 6 multipliers, 3 variable DFG's, and half of a resolver. The way in which the resolver is split up between two modules is discussed in detail in Chapter 7.

Summing up, I am not absolutely sure what the optimum module size is, but the above considerations indicate that a 680-sized machine should have between four and eight modules. The exact figure six was chosen for reasons of divisibility, as explained above. Since this module size satisfies criteria a), b), and c) as outlined above, it is probably fairly close to optimal.

The complement of other types of equipment in the module was obtained in most cases by dividing the normal 680 complement by six. The multiplier complement was increased slightly beyond the normal 680 complement to account for the heavy use of multipliers in aerospace problems.

The summers deserve some special consideration. Most problems require somewhat fewer summers than integrators. This fact is obscured on many machines by the fact that some of the amplifiers supplied as "summers" are actually used as output amplifiers for nonlinear equipment, or as inverters. Since the proposed system assumes committed amplifiers for nonlinear equipment, the need for summers is reduced.

The proposed system actually uses more summers than the standard 680, because of the occasional need for summers with a large number of inputs. Suppose, for example, a six-input summation is required. Since the system has no six-input summers, it will be necessary to use two three-input summers and one two-input summer. This procedure is somewhat contrary to the reasoning in Section 6.2 discussing Proposition a). The three summers together contribute 8 inputs and 3 outputs to the switching matrix rather than the 6 inputs and one output required by a single summer. However, the use of modularization alters the picture. If the three amplifiers are in the same module, they may be connected together through that module's internal matrix. The external matrix will then have only the same six inputs and one output

that a single summer would have. However, the three summers are more flexible than one six-input summer, since they can be used separately in problems not requiring six-input summation.

Another point in favor of eliminating summers with many inputs is that whenever four or more variables are to be combined, they are almost certain to come from several different modules. Suppose, for example, that three variables generated in module 1 are to be summed with three variables generated in module 2, and the sum is needed as an input to a component in module 3. If a six-input summer within module 3 were used, all six variables would have to be trunked between modules. The number of inter-module connections could be reduced to four by locating the six-input summer in module 1 or 2. However, one can reduce the inter-modular traffic even more by using more summers with fewer inputs. If the three inputs are summed within their respective modules, then only two variables need be sent from module to module through the external matrix: the partial sums are generated with modules 1 and 2 and summed in module 3.

An even better approach is to sum three variables within module 1, trunk the result to module 2, combine it with the three other variables by means of a four-input summer within module 2, and trunk the sum to module 3. Only two summers are needed, and only two signals are sent between modules.

The balance of different kinds of summers in the proposed system was determined as follows: First, the problems furnished by NASA were analyzed to obtain distribution functions, i.e., plots of the number of summers required versus the number of inputs. I had performed a similar study in 1965 in connection with the 680 design, and it was gratifying to note that the NASA problems matched the distribution obtained from the previous study. This distribution dictated the number of two-input summers, three-input summers, and so forth, that the system should contain. Then all summers with five or more inputs were replaced by combinations of summers with fewer inputs, in accord with the reasoning given above. The same process was followed with the integrators, with additional summers added to compensate for the elimination of five-and-six-input integrators.

The resulting summer complement was rounded off to the nearest multiple of six and divided into six equal modules. This comple-

ment of equipment has worked out fairly well in actual programming. This explains why there are more summers than integrators in the proposed system.

The above discussion of the various alternative means of summing six inputs ignores the sign inversion associated with summation or integration. If one uses a six-input summer, there is a sign inversion. If one uses a pair of three-input summers followed by a two-input summer, there is no sign inversion, since each signal passes through two amplifiers. If a three-input summer and a four-input summer are used, three inputs are inverted and the other three are not.

If the inputs to the summer are from nonlinear components (multipliers or variable DFG's), then the sign of the input can be chosen arbitrarily, since one may obtain either sign by appropriate programming. In loops that are purely linear, additional inverters may be necessary.

6.5 Design of the External Matrix

This section covers the design of the external matrix for a 680-sized system. The design is strongly influenced by the density of traffic expected on this matrix. We have already seen that more than half of the connections should be made internally, so that the external matrix may be designed for relatively light traffic.

The detailed structure of a module is covered in Chapter 7. For the present purposes, we need only note that each module has 25 component outputs and 62 component inputs. Thus the entire system (6 modules) has 150 component outputs (matrix inputs) and 372 component inputs (matrix outputs). Several different possible designs based on these figures are given below.

6.5.1 Rectangular Matrix

For comparison purposes, the switch count for a 150-by-372 rectangular matrix may easily be calculated. It requires (150)(372) = 55,800 switches.

6.5.2 Three-Stage Matrix

Suppose a single three-stage matrix is used (no modularization). With N = 150, M = 372, the formulas of Chapter 5 yield m = 14,

n = 8. The values of N and M must be increased to make them divisible by n and m respectively. Hence we need 19 input blocks, with 8 inputs per block (a total of 152 matrix inputs) and 27 output blocks, with 14 outputs per block (a total of 378 matrix outputs). Fourteen middle blocks are needed, and the total switch requirement is 14,602.

6.5.3 Modularized Array

We have seen that a modularized design, if it is to do any good, must be based on the assumption that only a fraction of the component inputs and outputs need to use the external matrix at any one time. The actual percentage is best determined by programming problems on the proposed modules.

The procedure followed was to draw the analog circuit diagram in the conventional way and then "section" it visually by drawing lines on the diagram marking the borders between the modules. The procedure is largely intuitive, and I don't have an algorithm for it, but it isn't too difficult to carry out in practice. Of course, analog programmers often divide a diagram into sections anyway, just to facilitate analysis, and this procedure is similar except for the need to match the number of components in any section with the complement of components available in a module.

With proper sectioning of the problem, it turns out that only about 5 or 6 out of the 25 component outputs and 10 or 12 of the component inputs in a module need access to the external matrix at any time. Let us include a little safety margin and allow 8 component outputs and 16 component inputs per module to access the external matrix. The main matrix then has 48 component outputs (matrix inputs) and 96 component inputs (matrix outputs).

Using the formulas of Chapter 5 with N = 48, M = 96, we get m = 8 and n = 5. Since 48 is not divisible by 5, we increase the number of inputs to 50, and we obtain 10 input blocks of 5 inputs each, 12 output blocks of 8 outputs each, and 8 middle blocks, each of which is a 10-by-12 matrix.

The total switch count for this matrix is 2,128, which is satisfyingly small, in comparison with the figures obtained for the other matrices. Of course, we must include the switches in the six internal matrices. However, the internal matrices are small, and if the total switch count in these six matrices is of the same order of magnitude as the

switch count in the external matrix, then the system will use fewer switches than the single large matrix discussed in Section 6.5.2.

The proposed system allows one-third of the component outputs to have access to the external matrix, but which ones should they be? On the principle that all modules should be identical, we should allow external access to two of the six summers, two of the six multipliers, one or two of the five integrators, one of the three DFG's, and so forth, in each module.

Any attempt to program problems on such an arrangement turns out to be hopeless. The difficulty is that although fewer than one third of the component outputs need external access in any given problem, the requirements differ widely from problem to problem and from module to module within a problem. Eight component outputs out of 25 are almost always more than enough, but the requirement in one module may be for 4 integrators, 3 summers and 1 multiplier with external access, while another may need 2 DFG's and 4 mulitpliers with external access. A similar comment holds for component inputs.

Thus, even though only about one component output in three will use the external matrix in a given problem, all outputs should have access to it.

The telephone company faces this same situation: at any given time only a small fraction of the telephones are in use, but every telephone must have access to every other (cf. Section 3.4). They solve the problem by using concentrators, and we can do the same thing.

6.5.4 Modularized Array with Concentration

Consider the 50 by 96 three-stage matrix designed in Section 6.5.3. Each of the 150 component outputs should have access to the 50 matrix inputs, and each of the 372 component inputs should have access to the 96 matrix outputs.

To achieve this goal, we may insert a 150 by 50 concentrator matrix between the 150 component outputs and the 50 matrix inputs. Similarly, a 96 by 372 expander may be inserted between

the 96 matrix outputs and the 372 component inputs. This arrangement provides the needed flexibility. Although only 50 of the 150 component outputs can use the external matrix at any one time, all have access to it. The output expander performs a similar function for the 372 component outputs. (The phone company would probably refer to both matrices as "concentrators," since they are not concerned with fan out, and hence need not distinguish between inputs and outputs).

Although this system has the necessary flexibility, it uses almost as many switches as the 150 by 372 rectangular matrix described in Section 6.5.1, and more switches than the 150 by 372 three-stage matrix described in Section 6.5.2. The concentrator matrices use far too many switches to be practical. Although the principle of concentration is sound, it must be implemented in some other manner.

6.5.5 Combining Concentrators with Input Blocks

The price of concentration can be greatly reduced by combining the input concentrators and output expanders with the input blocks and output blocks of the three-stage matrix. The 50 by 96 three-stage matrix has 10 input blocks and 12 output blocks.

If we enlarge the input blocks to accommodate 150 inputs, we obtain the necessary input concentration at a modest cost in switches. The input blocks must have 15 inputs each (15 x 10 = 150). The same trick applied to the output block results in 31 outputs on each of the 12 output blocks (31 x 12 = 372).

The net result is a three-stage matrix with 10 input blocks (each of which is 15 by 8), 8 middle blocks (each of which is 10 by 12), and 12 output blocks (each of which is 8 by 31). The total number of switches is 5,136. Although this is considerably more than the 2,128 required by the system in Section 6.5.3 (which proved adequate), it is also considerably less than the 14,602 switches required by the system in Section 6.5.2 (which has the same number of inputs and outputs).

Note that although this matrix has 150 inputs and 372 outputs (and carries light traffic; only 1/3 of its inputs and about 1/4 of its outputs in use at any time), it was designed as a 48 by 96 matrix carrying heavy traffic. It was pointed out in Section 3.4 that the Bell

System enjoys the advantage of light traffic. In Section 6.2 it was pointed out that we should not attempt to design a light-density matrix which would allow only a small fraction of the components to be used. The design in this section does use a light-density matrix, but not at the cost of forcing a large percentage of the analog components to be idle. Components whose input and output terminals are not used in the external matrix are not necessarily idle; they may still be used internally.

It also appears that this is probably a good way to design a three-stage matrix for light traffic density: first, use the estimated maximum traffic density (in this case, 33% on inputs and 25% on outputs) to determine the maximum number of matrix inputs and outputs likely to be actually used in a given problem (e.g., 48 and 96); second, design the matrix for heavy traffic with this many inputs and outputs; and third, enlarge the input and output blocks to include all inputs and outputs that actually need access to the matrix at one time or another.

Note that enlarging the input and/or output blocks does not affect the middle blocks--neither their number nor their size. It has already been pointed out in Chapters 4 and 5 that the middle blocks are the most prolific users of switches in an optimized three-stage matrix. This fact offers a partial explanation for the fact that tripling the size of the middle blocks and quadrupling the size of the output blocks only increases the number of switches by about a factor of two. Although the input and output blocks in the main matrix carry fairly light traffic, we should expect the traffic in the middle blocks to be quite heavy.

The three-stage matrix described in this section is not intended to be adequate for implementing all possible programs in the sense of Chapter 2. In fact, any program that uses more than 8 of the 15 inputs on a single input block or more than 8 of the 31 outputs on a given output block obviously requires more than 8 middle blocks. In fact, the matrix is not even adequate for all programs using 48 or fewer inputs and 96 or fewer outputs. A program might use only 75 matrix outputs, but if more than 8 of them are on the same output block, the program cannot be implemented with only 8 middle blocks.

In order for a program to be implemented on this matrix, two conditions are necessary: first, that the total number of inputs

and outputs in use should not be too great, and second, that they be distributed approximately evenly over the input and output blocks. The first condition has been met by determining the module size to reduce the traffic on the external matrix (allowing most connections to be made internally). The second condition can be met by proper assignment of components, if the matrix is suitably designed (see Section 6.7).

6.6 Input Blocks, Output Blocks, and Modules

The component outputs terminate in the matrix input blocks, and the component inputs terminate in the matrix output blocks. We have not yet decided how these blocks are to be related to the modules. Two questions present themselves:

a. Should all component outputs within a module terminate on the same matrix input block, or should they be scattered over all input blocks? It turns out that the optimal matrix design calls for more input blocks than modules; hence the question should be amended to read "should the component outputs in a module terminate in the minimum number of input blocks, or should they be spread out over the maximum number of input blocks?"

b. Should a given input block terminate many similar components or a balanced complement of different types of components?

Note that we really have four questions here, since we must answer these two questions for both input and output blocks.

The answers to this question are partly constrained by divisibility considerations. For example, the recommended design for the external matrix uses 10 input blocks and 12 output blocks. The 12 output blocks allow an easy division (two per module) while the 10 input blocks do not. However, it turns out that both inputs and outputs within a module should actually be spread out fairly widely over the matrix output and input blocks, so that these divisibility considerations are not crucial after all.

Consider first the output blocks. We have already seen (Chapter 5) that it is desirable to have fanout occur within an output block, rather than an input block or a middle block. The greater the output block

fanout, the fewer entries in the programming array, and the greater the chances of successfully implementing the program on the switching matrix. For example, if 25 component outputs (matrix inputs) are using the external matrix to connect to 50 component inputs (matrix outputs) then the number of entries in the programming array can vary from 25 to 50, depending on output block fanout. If all fanout takes place within output blocks, then each row has just one entry, and there are only 25 entries altogether. If none of the fanout takes place within output blocks, then there will be 50 entries.

The output blocks should be arranged to increase the likelihood of output block fanout. We have already seen that this implies that no summer, integrator, or multiplier should have two of its inputs on the same output block, since there is no reason to feed the same variable into both inputs on one component.

An additional contribution to output-block fanout is made by observing that large-fanout variables usually feed many components of the same type, rather than many different components. For example:

a. In aerospace problems, there are a few variables such as Mach number and altitude that drive many DFG's.

b. In many problems, there are time-varying co-efficients that are common to several equations; such variables must feed several multipliers. Such co-efficients often occur in applications involving the adjoint technique or Pontryagin's Maximum Principle, but also occur in other contexts; a variable-frequency sinewave oscillator is an example.

c. In partial differential equations, because of their symmetry, many variables will feed several summers or several integrators.

d. In harmonic analysis it is often desired to feed the same signal into several tuned filters with different transfer functions; this means that the signal will fan out to many integrators or summers.

In short, if a component has a high fanout, it is much more likely to feed many similar components than it is to feed a "balanced"

sample of the available components. We can take advantage of this fact by grouping many similar component inputs on the same matrix output block. If, for example, we terminate all variable DFG's on one output block, then a variable (such as Mach number) can drive many DFG's and still have an apparent fanout of one in the programming array (only one entry in its row). This extreme is probably not desirable, as it would limit the access to the DFG's in programs where they were all driven by different variables (e. g., P. D. E. 's with temperature-dependent functions generated within each cell). The proposed "large system" (2 680's) puts all 36 variable DFG's on 4 of the 16 output blocks (although they could have fitted on two, since there are 20 outputs per block). With this arrangement, a variable can drive 10 or 12 DFG's and still have an effective fanout of only 4 at most. Careful assignment of components should reduce this figure to 2 or 3.

In addition, if a variable feeds several similar components, it is fairly likely that they will be on the same module (e. g., the frequency factor in a sineware oscillator, or the aerodynamic co-efficientsin an aerospace simulation, which are concentrated within a few computing loops). Hence all three DFG's within a module appear on the same output block.

With fixed DFG's, the situation is different. There is no need to generate X^2 or sinX or log X more than once, so that inputs to similar DFG's should be "spread out" over as many output blocks as possible.

As for input blocks, the major consideration is to arrange the terminations to distribute the load as close to equally over the input blocks as possible. This design problem is linked to the problem of an algorithm for assigning components. If there are very many inputs in one input block in use (especially if some of these have high fanout) then many middle blocks will be required. To relieve this problem, one can interchange similar components on different input blocks. For example, suppose one input block has too many inputs in use while another has very few. Suppose integrator A is terminated on the "heavily loaded" input block and has high fanout, while integrator B is terminated on the "lightly loaded" input block, and has a fanout of zero. (This does not mean that integrator B is not used; it might connect to other components within its own module, but is does not use the

external matrix.) If we interchange integrators A and B, then we transfer one row from the heavy input block to the light input block, and thus even the load. To allow us the maximum freedom in doing this, we should not put many identical component outputs on the same input block. Thus each input block, unlike the output blocks, should terminate a balanced assortment of different types of components.

Also, this swap will not gain very much unless A and B are in the same module. We were assuming that B used only internal connections and hence had no entries in its row on the programming array. If integrators in different modules are swapped, then the connections that were internal become external, so that both rows of the programming array have entries, and we have not succeeded in reducing the number of inputs in use on the "heaviest" input block. Thus, priority should be given in the algorithm to those exchanges that can be made between components within a module. This answers question a): Identical components within a module should be spread out over as many input blocks as possible. (This also is in direct contrast to the situation for output blocks.)

6.7 Assignment of Components

Any algorithm to assign components should have two main parts: a sectioning routine for dividing the problem into sections (corresponding to modules) and a routine for assigning components within a module. The sectioning problem is fairly tricky; no attempt has been made to write such a subroutine in this project. Sample problems have simply been sectioned "by eye" after careful examination of the block diagram. This turns out to be fairly easy to do, but it would probably take a considerable amount of work to reduce it to an algorithm. Such an algorithm should be included in any operating system, but I don't think it's necessary for a feasibility study.

The second part of the assignment algorithm is much more straightforward. This routine has not actually been written, but the general outlines are clear. In fact, this problem had to be considered to some extent in order to answer the questions in Section 6.6.

The best approach at present seems to be iterative. Assignments are initially made in a "random" manner (although if any rules are available for determining a good starting point, they should be incorporated). Once the assignment is made, the programming array

is generated, and examined to determine the number of middle blocks required, and, more importantly, where the worst blocking occurs. The number of middle blocks cannot be less than the maximum number of entries in any column (the maximum number of outputs in use on any output block). It also cannot be less than the maximum number of inputs in use on any input block, although it will be greater than this if several of these inputs have high fanout and hence must fan out within the input block.

Hence we should start by determining the worst input and output blocks as follows:

a. Count the entries in each column and note the column with the most entries. This is the worst output block.

b. Count the inputs in use in each input block. Inputs with high fanout should count more heavily than inputs with low fanout. Hence a set of weighting factors should be introduced. These weighting factors should be determined empirically, but approximate values can be estimated a priori. According to the theory developed in Chapter 5, the input blocks should be approximately 2-to-1 expanders if the matrix is large; which allows each input to fan out to two middle blocks. Examination of several hundred programs (see Chapter 9) indicates that it is quite rare for an input to fan out to more than two middle blocks, even in "worst case" programs. Hence an input with a fanout of one should have a weighting factor of one (indicating that it will require one middle block) and the weighting factor should gradually increase to a limiting value of two as the fanout increases, indicating the increasing probability that it will require two middle blocks. As a starter, we may take a weighting factor of 1.5 for an input with a fanout of two, 1.75 for a fanout of three, and so on, with each additional fanout contributing only half as much as the preceding one. The sum of all such weighting factors in a given input block is the probable number of middle blocks required by that input block, and the maximum of these numbers determines the "worst" input block.

Once the worst input block and output block are determined, they are compared, to determine which is worse (i.e., which has the heaviest load, and hence which contributes most to the need for

middle blocks). Suppose, for example, the worst input block is the culprit. Attention is now focused on this input block and the various opportunities for swapping components to relieve this blocking are examined. Since each input block contains a balanced assortment of multipliers, DFG's, summers, and integrators, and since each of these components is interchangeable with several similar components which are within the same module, but on different input blocks, there are many opportunities for interchanging some of these components with similar ones on other input blocks which do not require external access. Each such swap will interchange two rows of the matrix and shift part of the "load" from the most heavily used input block to some other, less heavily used input block.

After each swap, steps a) and b) should be repeated to see if the situation really has improved, and if a different input block (or output block) is now the worst. The swapping should continue until several successive swaps fail to bring about improvement; when this happens, it means that there are several different "worst" input and output blocks, all approximately equal, so that the load is more evenly distributed.

Of course, if two component outputs are interchanged to relieve congestion within an input block, then it is also necessary to interchange the inputs to these components as well. This will have some effect on the output blocks, but it will probably not make the worst output block any worse. For example, suppose two variable DFG's within a module are interchanged. Since variable DFG's within a module have their inputs on the same matrix output block, the interchange of inputs will not affect the programming array. Even if two components with inputs on different output blocks are involved in the swap, the probability that the worst output block is one of the two involved in the swap is fairly low (unless there are several different equally heavily-loaded output blocks).

If the worst output block is the limiting factor, then many interchanges can be made without disturbing the worst input block. For example, we have already seen that two inputs to a multiplier or a summer or integrator should be on different output blocks. These may be swapped at will without affecting the output of the component. (The software must be sufficiently sophisticated to realize that this can be done with a multiplier only when it is in the MULTIPLY mode, not when it is in the DIVIDE mode.) In any column with many entries,

the majority of these entries will be interchangeable with terminals in some other column; by interchanging two such connections, an entry is transferred from one column to another. Thus, exchanges can be found which will transfer entries from the worst column to others less heavily loaded (unless the load is evenly distributed already).

7. DESIGN OF THE ANALOG CONFIGURATION: SPECIFIC DETAILS

This chapter describes the suggested systems, including the analog components and the switching matrices. Two systems are proposed; a small system (one 680) and a large system (two 680's).

7.1 Configuration Switching

The proposed components are fairly complex in that they require a lot of configuration switching. I have already pointed out in Section 6.2 that this is desirable, but the question still remains whether all components of a given type need configuration switching. Must all multipliers be capable of division, for example? In a given problem, less than half the multipliers will be used as dividers, so that we could specify that only 50% of the multipliers have division capability and save some configuration switches. The same holds true for sign inversion; we might eliminate the relays that allow interchanging + and - inputs on a multiplier. If the multipliers were hard-wired so that half of them had sign inversion and the other half did not, then we might be able to satisfy our programming requirements with proper assignment of components.

The reason for allowing so much configuration switching is to preserve interchangeability of components. We have seen in Sections 5.2 through 5.4 that many middle blocks are needed when many inputs are concentrated in one input block (especially if they have high fanout) and it is also obvious that heavy traffic on a single output block will also increase the number of middle blocks needed. In Section 6.7 an algorithm was outlined for distributing the "load" evenly by interchanging components. This will only work if there are sufficiently many interchangeable components. If one multiplier has division capability and the other does not, then they are no longer interchangeable. The proposed single-680 system uses only about 1000 configuration switches. The saving of switches in the switching matrix itself more than makes up for these configuration switches.

7.2 Internal And External Access

We saw in Section 6.5.5 that it does not cost too much to allow every component input and output to have access to the external matrix provided they don't all use this access in any single problem. Each component output in the proposed system has access to both external and internal matrices, but it turns out to be desirable to allow some of the component inputs to have access to one or the other, but not both. There are two main types of components for which this should be done:

7.2.1 Linear Components

Consider a summer with two inputs, both of which have access to both the internal and external matrix. Such a summer contributes two matrix output terminals to the external matrix and two to the internal. Now consider a summer with four inputs, two of which have external access only and two of which have internal access only. Both summers contribute the same number of terminals to both matrices, but the second one can do everything the first one can and much more. Hence every linear component (summer or integrator) in the proposed system is provided with at least one internal and one external input, and no inputs with access to both.

7.2.2 Components With Inputs Only (Or Outputs Only)

Special consideration should be given to components with no analog outputs and components with no analog inputs. In an analog system there are a small number of "output only" signals: plus reference, minus reference, ground, and a noise generator. There are many more "input only" devices, namely comparators, (which have no analog output) and all readout devices (plotters, scopes, strip-chart recorders, etc.). In a hybrid system, one should include A/D and D/A conversion channels as examples of "input only" and "output only" devices (although multiplying DAC's are an exception; they do have analog inputs). Trunks, of course, fall in the "input only" or "output only" category. No trunks per se have been included in this system, since a problem requiring two consoles may be solved more efficiently by simply enlarging the external matrix. As far as the programming of the switching matrix is concerned, it makes no difference whether two modules are physically in the same console or two different consoles. Of course, the inter-module trunking by means of the external matrix is analogous to inter-console trunking in a conventional analog system.

It appears that "input only" and "output only" components should be accessed only through the internal matrix. For example, compare an "input only" component such as an A/D channel, with a device like a DFG, which has an input and an output. Suppose I have the variable x generated in module 1 and I need f(x) as an input to a component in module 5. I can locate the DFG in either module 1 or 5, and send only one signal between modules. If I run out of DFG's in modules 1 and 5, then I can generate f(x) in some other module, but then I must send two signals between modules. A two-input-one-output device (such as a two-input summer or a multiplier) is best located in the sending module rather than the receiving module (If I want to multiply two variables that are generated in module 1 and use the result in module 5, it is better to locate

the multiplier in module 1). With an "input only" component like an A/D channel, there is no "receiving module", so that one can avoid trunking the signal by using an A/D channel in the same module as the one in which the variable is generated. Of course, one may run out of A/D channels within a given module, thus making it necessary to send the signal to some other module for conversion. This indicates that it is probably a good idea to have some channels in each module with both external and internal access, and some with internal access only.

However, if the "input only" components are sufficiently cheap, it is probably a good idea to reduce traffic on the external matrix by allowing such components to have internal access only and supplying slightly more than are likely to be needed. This was done in the proposed system with the comparators (which are fairly cheap) and the readout lines (which are very cheap; see Section 7.6.8).

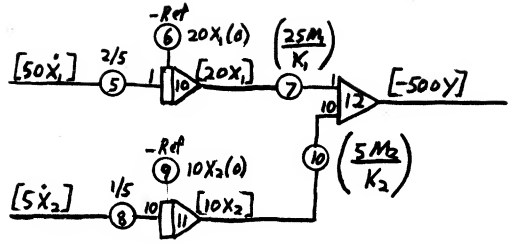
The same sort of reasoning applies with "output only" devices; there seems little point in sending plus or minus reference over the external matrix. Since reference is assumed to be supplied to every IC pot and every comparator bias pot, the only other use for reference is to feed summers and integrators. It is assumed that every internal input to a summer or integrator has access to both plus and minus reference. This adds only 38 switches per module. Furthermore, since each input has a committed pot (or the equivalent in weighted resistors), half of these switches might be the same ones that are needed to put plus reference on the pot for setting purposes anyway.

7.3 Committed Pots

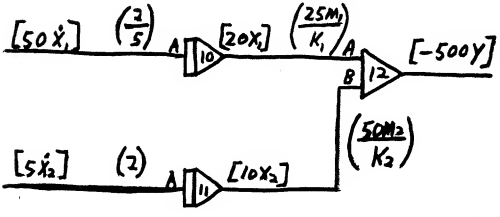
Each summer or integrator input in the proposed system has a pot committed to it. This committment is justified in Section 6.1 by comparing the cost of the extra pots with the cost of the switches saved. It should also be pointed out that the extra pots affect programming as well. The effect is a beneficial one, for at least two reasons. First, the pot need not be drawn on the circuit diagram, which not only saves effort, but reduces clutter, and second, the programmer need not separate a coefficient into a "pot-setting" and a "gain of 1 or 10", but can simply think of each input as having an adjustable gain, varying between 0 and 10. These adjustable gains can be addressed, set, and read just like a pot-setting.

As an example, consider the two diagrams in Figure 7-1. Both show the same scaled program for the equation

$$Y = \frac{M_1}{K_1} X_1 + \frac{M_2}{K_2} X_2$$



a) Conventional Program



b) With Committed Pots

Fig 7.1 Effect of Committed Pots on Programming.

Figure 7-1 a is programmed in the conventional manner, with separate pots. Each pot is labeled with an address (e. g. PO5, P10) and with an expression for the setting which may be either numerical (e. g. PO5 = 2/5) or algebraic (e. g. PO7 = $25M_1/1K_1$). Pot-settings greater than one must be divided by an appropriate value (usually 2, 5, or 10) to make them less than one, and the diagram must be modified to include this factor as an input gain to the amplifier (e. g. P10). The product of the pot-setting and the amplifier gain is what counts.

Figure 7-1 b shows the same scaled diagram with committed pots. Since each input has an adjustable gain, there is no reason to draw the pots explicitly. The inputs are labeled with the co-efficients just like the pots in Figure 7-1 a. For setup and readout, the co-efficients may be addressed with the number of the amplifier and a letter distinguishing among the various amplifier inputs (e. g. 12A is set to $25M_1/K_1$). The co-efficient can have any value between zero and a maximum value of about 10 or 20. The fact that an IC pot is committed to each integrator eliminates the need to draw and label the pot on the diagram. The IC pot may be addressed with the letter I, (e. g. 10I for the IC pot on integrator 10), which maintains the number-letter format similar to amplifier gains (e. g. 12B, 11A). The integrator IC's (and also comparator bias pots) can have negative values, unlike ordinary pot-settings.

Note the relatively uncluttered appearance of Fig. 7-1 b due to the elimination of pots. Equally important, although more subtle, is the fact that scaling is simplified because an input co-efficient does not have to be factored into a pot-setting and an amplifier gain.

7.4 Description of a Module

Figure 7-2 indicates the complement of equipment in a module of the proposed system. There are 25 components with addressable analog outputs and several additional addresses for "input only" components. Each component output has access to both internal and external matrices; component inputs can have internal access, external access, or both. Inputs which terminate in horizontal lines are external. Thus, component number 5 is a four-input summer, of which three inputs (B, C, and D) are internal, and one (A) is external. Component number 15 is a multiplier with two inputs, each of which has access to both internal and external matrices. Note that different letters are used for the same input, depending on whether internal or external connections are to be made. This double-labeling was done to simplify the digital program for assigning components (see Chapter 10).

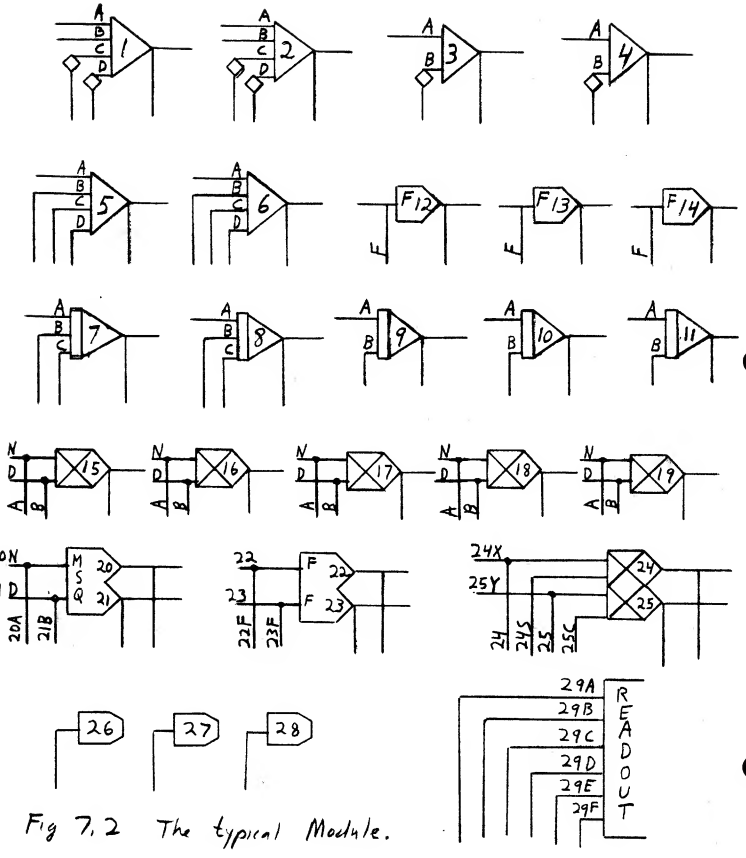


Fig 7.2 The typical Module.

The letters for component inputs were chosen to be mnemonically meaningful for the programmer whenever this was possible; where it was not, the letters A, B, C, etc. were used arbitrarily. For example, the inputs to an integrator or summer are simply labeled A, B, C, etc. For a single-input device such as a DFG, there is no need for a letter to distinguish the input, but in order to label the internal and external inputs differently, letters must be used. DFG number 12, for example, has a single input, referred to as 12F if it is to be connected internally, and simply as 12 if it is to be connected externally. Similar remarks hold for inputs such as 24X and 24, which represent the same input to the same component, but the former is external and the latter internal.

This section provides a brief summary of the components available in a given module; more detail on each component is given in Section 7.6.

Components 1 through 6 are summers. The output is minus the sum of the inputs, with each input multiplied by an adjustable gain. Note that the first four summers have logically-controlled D/A switches associated with them; it is assumed that these switches can be left permanently on, so that the inputs can be used as ordinary unswitched inputs to the amplifier when not otherwise in use.

Components 7 through 11 are combination integrator/summers. All are assumed to have built-in IC pots (not shown in Figure 7-2). It is also assumed that each integrator has an additional input (with adjustable gain) which can be connected only to its own output. This capability was included because many applications require adjustable damping co-efficients. This connection is sufficiently common to justify including it in each integrator, without adding additional inputs or outputs to the switching matrices.

Components 12, 13 and 14 are variable function generators, generating arbitrary functions of a single input variable. They may be either DFG's or hybrid function generators using logic and multiplying DAC's.

Components 15 through 19 are multiplier/dividers. Each is capable of either multiplication or division, and each has an adjustable gain, which is desirable for multiplication and essential for division; see Section 7.6.3 for more details. Each multiplier/divider is also assumed capable of being programmed for sign inversion or not, at the programmer's option.

Inputs to multipliers are interchangeable if the component is in the MULTIPLY mode, but in the DIVIDE mode they must be distinguished

by "N" for numerator and "D" for denominator. Hence these letters are used to identify the inputs (even in the MULTIPLY mode).

Components 20 through 25 are different from the others, in that they have two outputs.

Components 20 and 21 form a MSQ (multiplier/squarer). This is a multiplier which can be separated into two square or square-root circuits. In the MULTIPLY mode, the inputs 20N and 21D are multiplied and the result appears at output terminal number 20. Note that the input addresses use different numbers as well as different letters, because in the dual squaring mode, they are thought of as inputs to different components.

Components 22 and 23 form a dual function generator. Functions such as the loganthm, exponential, square and square-root, sine and cosine may be installed at the customer's option. For customers wanting resolvers, a dual sine/cosine unit may be installed. Since generating the sine and cosine of the same angle is a very common application, an internal relay is assumed which connects the two inputs together, so that the sine and cosine of a given input may be generated with only one external input connection.

One additional characteristic of the dual sine/cosine generator that distinguishes it from other DFG's is the internal switching that allows it to operate on all values of the input, and not merely values between -180^{o} and $+180^{o}$. Integrator #11 is assumed to have the necessary switching capability to enable it to operate in conjunction with sine/cosine unit 22/23 in rate-input resolver modes.

By long-standing tradition, a 680-sized computer is assumed to have three resolvers. Hence one resolver for every two modules is assumed. The modules are numbered 1 through 6 (1 through 12 in the "two-console" system) and every odd-numbered module is assumed to have a dual sine-cosine unit in this position. Each even-numbered module is assumed to have an additional multiplier/squarer in this position. Hence the total multiplier count consists of 30 multiplier/dividers (5 per module) and 9 MSQ's (one per odd-numbered module and two per even-numbered module).

Components 24 and 25 form the other half of the resolver: the multipliers. If resolver operation is not desired, these components may be used as free multipliers, in which case the output of component 24 is the product of the 24X and 24S inputs, and a similar statement holds for component number 25. In this mode of operation, the 24/25 components may be thought of as a pair of free multipliers, and all modules are identical in this respect.

For the axis rotation mode and the rectangular-to-polar mode, we must generate the expressions

$$U = Y \cos\Theta - X \sin\Theta \quad \text{and} \quad V = X \cos\Theta + Y \sin\Theta$$

These outputs appear at terminals 24 and 25 respectively. Note that these modes of operation require four multipliers, rather than two. In such modes, an odd-numbered module (such as module 3) will "borrow" the two multipliers (numbers 24 and 25) from the corresponding even-numbered module (e.g. module 4). In this case, the even-numbered module has two idle inputs and two idle output terminals, with amplifiers. To increase programming flexibility at relatively little cost, these amplifiers may be used as two-input summers when their multipliers are "borrowed" by the other module.

Components 26, 27, and 28 are electronic comparators, each with a built-in bias pot capable of any bias setting between - Reference and + Reference. Thus they are one-input-no-output devices.

Addresses 29A through 29F are for connections to readout devices (plotters, scopes, recorders, etc.). They may be considered interchangeable for programming purposes (see Section 7.6.8).

In summary, all six modules are identical except for the resolver components, which consist of components 22 through 25 and component 11 (in odd-numbered modules, this integrator contains the switches and logic necessary for continuous operation with angular rate input; in even-numbered modules, it is identical with the other two-input integrators). The total complement of equipment is as follows:

36 summers
30 integrators
39 multipliers (of which 9 have dual squaring/square-root capability)
3 resolvers (including 12 multipliers, which are not included in the 39 multipliers above)
18 electronic comparators
36 readout lines

7.5 Miscellaneous Components and Features

This section deals with components that are either ignored or simply taken for granted in the rest of this report.

7.5.1 Logic

Since logic signals and analog signals are never connected together, they form two separate switching problems. Since there are fewer

logic components than analog components, the logic matrix will be smaller than the analog matrix. The logic matrix also enjoys additional advantages in that it can use integrated-circuit gates instead of bulkier, more expensive mechanical relays. Noise and crosstalk should be much less of a problem than with the analog matrix. Hence it appears that if the analog matrix problem is solvable, then the logic matrix problem is almost certainly solvable. This report simply assumes that the needed logic signals are available to drive D/A switches, integrator mode control, etc.

Although this is a reasonable approach for a feasibility study, a few words should be said about the actual design problem. The straightforward approach of duplicating the complement of components on the logic panel of a conventional analog and interconnecting them by relays or logic gates is probably not the best. Consider, for example, such components as general-purpose logic gates. If logic gates instead of relays are used to interconnect these components, then the "components" become indistinguishable from the "switches" that interconnect them.

As another example, consider the memory elements in a logic system (flip/flops and various aggregates of flip/flops, such as registers and counters). Many of the functions of such elements are essentially sequential or stored-program functions that would probably be handled better by a GPDC. The original packaged logic systems were designed on the premise that GPDC's were expensive, and hence it was better to do as much as possible with the patched logic. Early logic systems contained such features as patchable binary adders and serial memory to allow arithmetic and storage functions to be carried out without a GPDC. Modern hybrid systems generally perform such functions by stored-program control.

It seems fairly clear that any automatic patching system must include a GPDC to handle the necessary setup calculations; fortunately, GPDC's today are sufficiently inexpensive to make this economically feasible. If such a system is available with the analog at run-time, it may also take over much of the control functions that would otherwise be patched with shift registers and counters. Under these conditions, the logic system might be reduced to a network of gates connecting comparator outputs to D/A switch and mode control inputs, with the GPDC changing the interconnections dynamically under stored-program control.

Although the desired logic connections may simply be assumed to exist for purposes of a feasibility study, the actual design of such a logic system is not a trivial task, and is worthy of a separate study if the analog hardware proves to be feasible.

7.5.2 Track/Store Units

Early track/store applications used integrators as the storage elements; modern machines generally build the track/store network into summing amplifiers so as not to waste the precision feedback capacitors and other expensive components within an integrator. This approach is probably desirable for an automatically patched system as well. Some modifications may be desirable. For example, track/store units are usually used in pairs. In view of the importance of having as few input and output terminals as possible, it might be desirable to construct a track/store pair from two amplifiers and treat it as a single component from the point of view of the relay matrix.

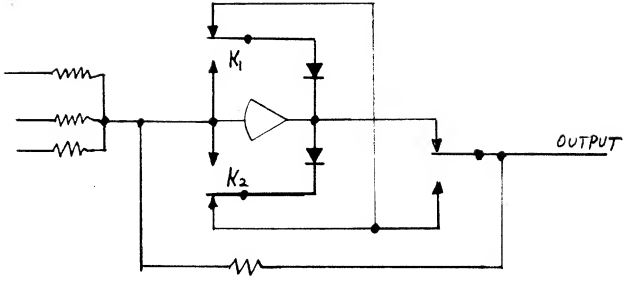
In fact, the inclusion of a digital computer as an integral part of the analog raises the question of whether it is not preferable to do most or all of the data storage digitally, thus eliminating or at least reducing the need for track/store units as such. If every A/D channel had a committed track/store unit and every D/A channel a committed analog interpolator, there would be less need for track/store units as separate components.

7.5.3 Hard-Zero Limits

The hard-zero limit has already been discussed in Section 6.2. At least two summers per module should have this feature, and perhaps it is even desirable to provide it on all summers to maintain a greater degree of component interchangeability. Figure 7-3 shows two possible schematics for such a feature. One requires three form-C switches and the other requires four form-A. In both cases, the switches are shown in the normal (non-limited) state.

7.5.4 Feedback Limiters

The feedback limiter is especially tricky because it requires a summing-junction connection to attach it to a specific amplifier, and summing-junction switching may lead to serious crosstalk and stability problems. Examination of applications for feedback limiters indicates that they fall into three categories. Sometimes it is necessary to establish an upper or lower bound on an analog variable; this is a true limiting application. Sometimes amplifiers are used with no feedback except a limiter; such an amplifier is always in one or the other of its limited states, and is therefore a binary device. Comparators, gates, and even flip/flops can be constructed in this way, but in a modern analog there is little need for such tricks; it is cheaper (and faster) to use packaged logic. The third use of a feedback limiter is as a "pacifier" to prevent overloads. Such an application does not require the limit to be adjustable.



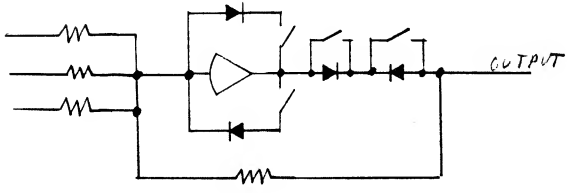


Fig 7.3 Hard-zero Limit on Summer

To avoid summing-junction switching, one may commit a feedback limiter to each amplifier that is likely to need one (principally summers and integrators). This seems very wasteful of equipment, but perhaps it is not as expensive as it appears. The actual limiting circuitry appears to be fairly cheap: a few resistors and diodes for a soft limit, and a few extra transistors for a hard limit. The main cost seems to be in the adjustment pot (which, in a system like this, should be a servo-set pot or perhaps a switched resistor network; at any rate, it should be digitally set).

In any given problem, only a few amplifiers will need limiters, but we don't know in advance which ones they will be, and we want to maintain as much component interchangeability as possible, for the reasons covered in Section 6.7. Suppose we commit a feedback limiter to every summer and every integrator (a total of eleven limiters per module) but share the adjustment pots. If we provide two pots for upper limits and two pots for lower limits in each module, then any two of these eleven amplifiers can be limited to independent upper and lower limits. More than two amplifiers can be limited if some are limited only in one direction, or if several amplifiers have the same limits - a fairly common occurrence.

Figure 7-4 illustrates the principle. Simple passive diode limiters are used for illustration, but active (hard) limiters could also be used. The total cost is 66 switches and 4 pots per module, not counting the limiter circuitry itself. No summing-junction switching is required. When an amplifier is not supposed to be limited at all, the switches set the limits to $\pm$ 12 volts ($\pm$ 1.2 units) which is outside the normal computing range. Hence the limiter will not affect normal operation, except for the beneficial effect of preventing hard saturation and improving overload recovery time. It should also be possible to improve the dynamic characteristics of the limiter if it is committed to a specific amplifier and has no long summing-junction leads.

7.6 Detailed Component Description

The proper way to define a component is in terms of what it does, rather than what's in it. Although schematics are provided for each of the components, these schematics are only suggestions. Different configurations of switches may perform the same analog function better or more economically; it is the functions themselves that count.

This section describes the analog components as if they were built from scratch. I am not sure how practical it is to implement these ideas on an existing computer. As pointed out in the introduction, it

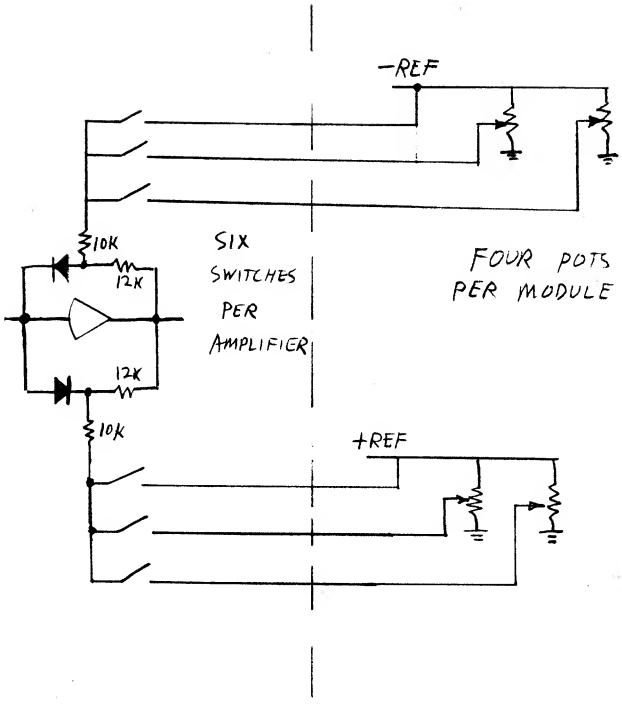


Fig 7.4 Feedback Limiter with Shared Pots

would almost certainly be necessary to make some internal modifications within the analog console, but I'm not sure how extensive this job would be.

7.6.1 Summers

Every summer has a committed pot for each of its inputs. Some economic justification has been given for this step in Sections 6.1 and 7.3, but there is no doubt that the committed pot approach is more expensive (in terms of analog components) than a simple input resistor. This section describes one method of reducing the cost somewhat. With committed pots, it is not really necessary to use precision resistors. In addition, some of the setup and readout relays can be absorbed into the switching matrix itself. It is assumed here that the pot is set through the stored-program capability of a GPDC, which has access to the matrix relays. The general idea is equally applicable if weighted input resistors, instead of pots are used. The proposed pot-setting procedure may also give better accuracy than conventional pot-setting techniques, since it is not dependent on network resistor accuracy.

Figure 7-5 illustrates the suggested schematic. None of the resistors shown need be very precise; 1% tolerance is adequate. When setting the pot, the digital computer temporarily disconnects the pot input from its programmed connection (using the regular matrix relays), and connects reference voltage instead. Note that either + or - reference may be connected to the pot. Of course, only one of these is necessary to set the pot, but both are provided because the actual programmed input may be + or - reference. The digital program also disconnects all other input pots from this amplifier, and then connects the amplifier output to the readout bus. This may be done with the regular readout relay for that particular amplifier. The pot is then driven until the amplifier output has the desired value (it is nulled against a precision voltage source, just as in conventional pot-setting). Even though low-accuracy resistors are used, the overall gain is correct. After a null is reached, the original problem connections are restored.

Note that the input resistor is 90K, allowing a gain of at least 1.1 with the 100K feedback resistor. If greater gains are required, the 10K input resistor may be switched in, allowing a gain of 11 or more. For setting large gains without overloading the amplifier, the reference input should be replaced by 0.1 times reference. This requires an additional amplifier and precision resistors, or some equivalent low-impedance voltage source, but only one such source is needed for the entire machine, not one per pot. The entire reference bus is tempor-

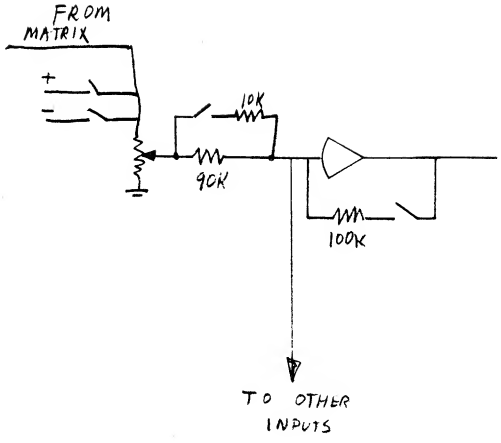


Fig 7.5 Summer with Pot-Set Failure

arily switched to this source for the pot-setting operation, and the precision voltage source containing the pot-setting (a precision DAC) is modified accordingly.

With this arrangement, the operator can enter (manually, or from tape or cards) any co-efficient between 0 and 11 and have it set to four-figure accuracy.

7.6.2 Integrators

Input pots on integrators may be set the same way as for summers. Each integrator is also capable of acting as a summer, and hence internal switching must be included to allow the feedback capacitor to be replaced by a resistor. This resistor may be switched into the circuit and the capacitor removed when the input pots are being set, so that the procedure becomes the same as for a summer. The feedback resistor value need not be precise, but all capacitors in the integrator should be trimmed to provide the correct RC time constant in conjunction with the integrator's own feedback resistor, since this resistor is used in setting input co-efficients.

To set the committed IC pot, the digital program forces the integrator into the IC mode and turns the pot until the desired value is obtained at the amplifier output. The IC feedback resistor should be slightly larger than the IC input resistor, allowing a nominal overrange of about 5%. This will assure that a setting of 1.0000 can be obtained despite resistor mismatch and pot end-resistance.

7.6.3 Multipliers

Figure 7-6 gives the suggested schematic for a multiplier/divider. A relay with two form-C contacts is used to select the MULTIPLY and DIVIDE modes, and another to select the output polarity.

The scaling relays allow the feedback resistor to have any value from 10K to 100K, in multiples of 10K, except for 50K and 60K. In the MULTIPLY mode, this feature allows more accurate generation of products of "out-of-phase" variables (that is, one input is very small when the other is large). One of the most common examples is dynamic pressure (ρv^2), where ρ is large and V small at low altitudes and the opposite is true at high altitudes. Although this scaling feature is desirable for multiplication, it is essential for division, where the "feedback" resistor becomes an input resistor. If the numerator is not attenuated, the quotient will probably overload.

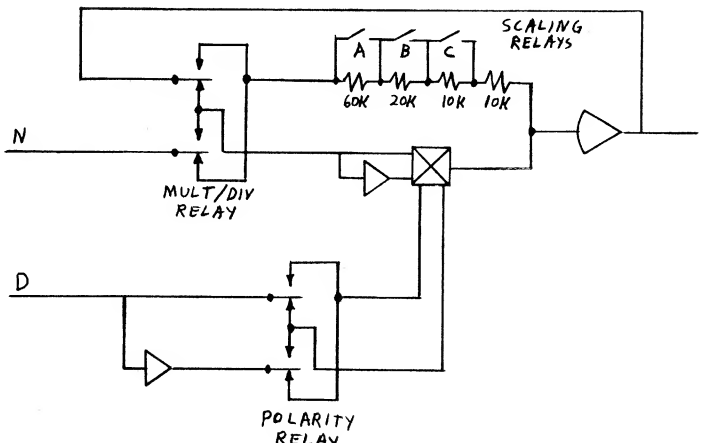

SCALING
RELAYS
A
B
C
60K
20K
10K
10K
N
MULT/DIV
RELAY
D
POLARITY
RELAY

Fig 7.6a

In the division mode, the programmer does not really have choice of output polarity. He must choose the polarity relay setting according to the sign of the denominator, to make the loop stable. There may or may not be a sign inversion in the division, but the programmer has no sign option.

This option may be added by means of a form-C relay at the output, as shown in Figure 7.6-b. The programmer (or the software) positions the input polarity relay according to the sign of the denominator, and then uses the output polarity relay to determine the sign of the quotient.

7.6.4 The Multiplier-Squarer

In the dual squaring mode, an MSQ requires four amplifiers. The proposed design incorporates a fifth to allow some freedom of output polarity choice. This is necessary because conventional quarter-square circuitry normally provides $-X^2$ and $+Y^2$ in the dual squaring mode. Interchanging + X and - X terminals to the squaring network does not change the sign of the output.

Usually, when two variables are squared, it is desirable to have the same sign available for both outputs (e.g. to generate $X^2 + Y^2$). If they had different signs, it would be necessary to pass one of the signals through an external inverter, which would tie up matrix switches and general-purpose equipment.

Relay K_6 allows the second amplifier to produce the inverted output in the MULTIPLY and DIVIDE modes. When used in the dual X^2 mode, Relay K_6 allows $X^2 + Y^2$ to be generated directly without an external summer.

7.6.5 The Dual Function Generator

This is a two-input-two-output device (or a pair of independent single-input-single-output devices). Log/exponential units or sine/cosine units may be terminated here. One could also terminate a dual squaring unit here, but if that is done, it appears desirable to go "all the way" and use a regular MSQ, so that it can multiply and divide as well.

For a six-module, three-resolver system oriented toward aerospace applications, this report recommends a dual sine/cosine unit in odd-numbered modules and an additional MSQ in even-numbered ones. The sine/cosine DFG is different from the other types of dual DFG's in several respects:

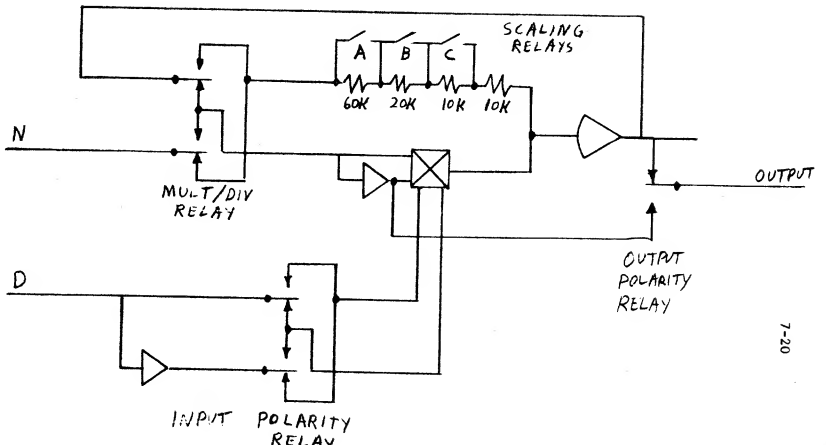

SCALING
RELAYS
A
B
C
60K
20K
10K
10K
N
MULT/DIV
RELAY
OUTPUT
OUTPUT
POLARITY
RELAY
D
INPUT POLARITY
RELAY

Fig 7.6b

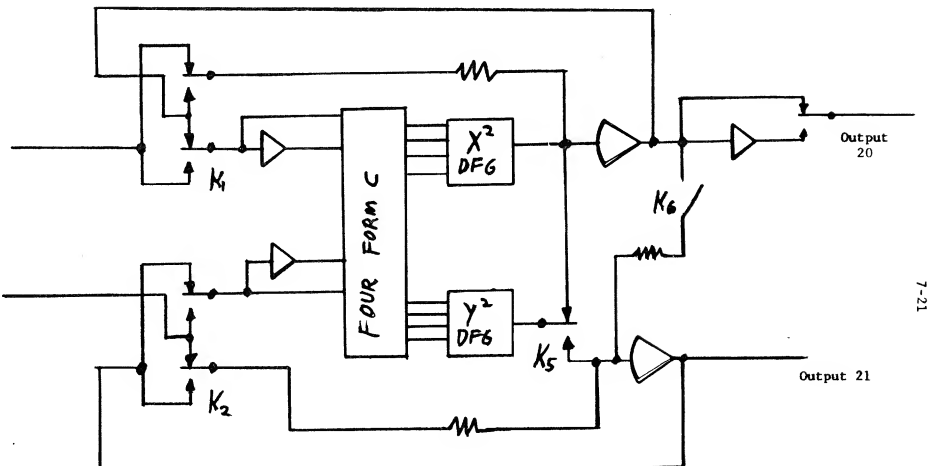


Fig 7.7 Suggested Schematic for Multiplier/Squarer

a) It must contain more amplifiers, since it generates a non-monotonic function.

b) It should contain comparators, logic, and switches to enable it to operate in conjunction with integrator 11 (in the rate input modes) and with amplifier 24 (in the rectangular-to-polar mode) to generate the correct values of $\sin\Theta$ and $\cos\Theta$ when Θ is unbounded. This circuitry may be a plug-in expension, or perhaps it should be built into the unit from the start.

c) An internal relay should be provided to connect the two inputs together under program control. This will allow the generation of $\sin\Theta$ and $\cos\Theta$ with only one matrix connection, rather than two.

7.6.6 The Resolver Multipliers

At first glance, the electronic resolver appears to be a desirable "black box" component with only a few inputs and a few outputs. For example, in the polar-to-rectangular mode, the inputs are R and Θ, and the outputs are X ($=R\cos\Theta$) and Y ($=R\sin\Theta$). In the inverse mode, the inputs are X and Y and the outputs R and Θ. The axis rotation mode requires three inputs (X, Y, Θ) and produces two outputs ($U = X\cos\Theta + Y\sin\Theta$ and $V = Y\cos\Theta - X\sin\Theta$). Thus it appears that the resolver should be terminated as a three-input-two-output device. This is consistent with the "black box" or "committed component" approach used in other components.

Examination of the actual uses of a resolver dispels this notion. More terminals must be added. For example, the "intermediate variables" $\sin\Theta$ and $\cos\Theta$ are generated within the resolver. Very often these variables are needed elsewhere. Also, it is often desirable to use the multipliers as free components, which requires additional input terminals. Hence the present design treats the resolver as two separate components: a sine/cosine unit and a quadruple multiplier.

The multiplier part terminates as a four-input-two-output device within each module. It has already been explained in Section 7.4 that a resolver in an odd-numbered module "borrows" the multipliers from the corresponding even-numbered module. Figure 7-8 shows how this may be done. The relays are shown de-energized, which is the proper state for the axis-rotation mode. By energizing relays K_1 through K_5, the multipliers are separated into free components.

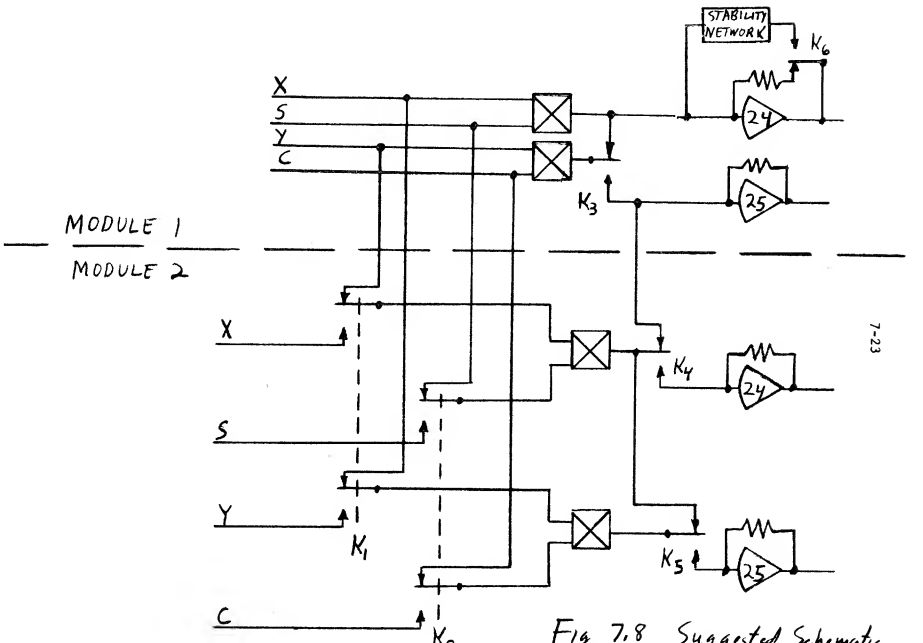


Fig 7.8 Suggested Schematic for Resolver Multipliers

Additional relays (not shown in Figure 7-8) perform the following functions:

a) Switch in input resistors to allow amplifiers 24 and 25 in module 2 to be two-input summers when their multipliers are "borrowed" by module 1.

b) Connect the "S" and "C" inputs to the resolver in module 1 to the sine and cosine. outputs (22 and 23) without going through the switching matrix.

c) Connect the "Θ" inputs on sine/cosine unit 22/23 to integrator 11 for rate modes.

d) Connect the X and Y inputs in both module 1 and 2 together (for the polar-to-rectangular mode).

Note that b), c), and d) represent connections that could have been made with the internal matrix within the modules, but these connections are so common that it is probably worth while to make them directly, thus reducing the traffic on the switching matrices.

Figure 7-9 illustrates connections to be made for the various modes. Dotted connections are not part of the switching matrix, but are made within the resolver.

In 7-9 a, the axis rotation mode, all relays are de-energized, as shown in Figure 7-8. Components 24 and 25 in module 2 are available as two-input summers.

In 7-9 b, the PR-1 mode, components 24 and 25 in module 2 are available as multipliers. Relays K_1 through K_5 are energized.

In Figure 7-9 c, the PR-2 mode is illustrated. Relays K_1, K_3, K_4, and K_5 are energized, but K_2 is not. Note that two resolutions with a common angle are being performed in different modules, but it is not necessary to trunk $\sin\Theta$ and $\cos\Theta$ through the external matrix, because K_2 is left de-energized.

In Figure 7-9 d (R-P mode), only K_6 is energized within the resolver, thus substituting a stabilizing network for the feedback resistor. In addition to the stabilizing network, this relay also connects the necessary logic and switching circuitry to assure proper operation when Θ goes beyond $\pm$ 180 degrees.

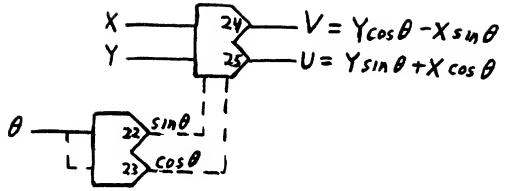


Fig 7.9a Axis Rotation Mode

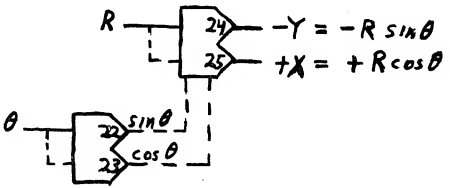


Fig 7.9b
PR-1 Mode

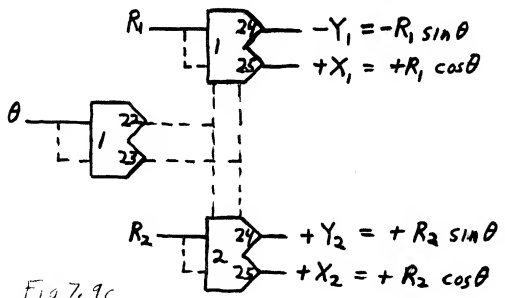


Fig 7.9c
PR-2 Mode

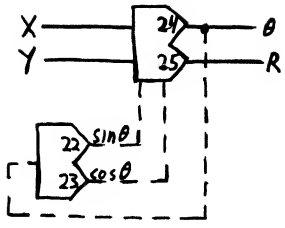


Fig 7.9 d. R P Mode

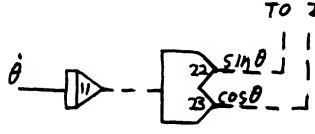


Fig 7.9 e. Rate Input Modes (PR-1, PR-2 or Rotation)

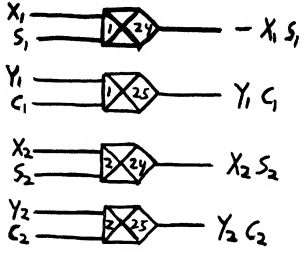


Fig 7.9 f. [illegible] Multiplier Mode

Figure 7-9 e illustrates the rate input modes. Connections for PR-1, PR-2, or axis rotation modes are made as usual, but the input to the sine/cosine unit is derived from the integrator 11, which contains the necessary switches and logic for continuous rotation.

Figure 7-8 f shows the free multiplier mode. Note that only multiplier 24 in the odd-numbered module has sign inversion. The signs were chosen for reasons of stability in the inverse mode. It may prove desirable to introduce division relays and polarity relays to allow these multipliers to be interchangeable with the standard multipliers, but I have not done so because it appears that these multipliers will be used in the resolver mode most of the time.

7.6.7 The Comparator

Although most comparators have two inputs (and the 680 comparator has three), the most common use is to compare a varying voltage against a fixed bias level. Hence a fixed bias pot is included, so that the comparator becomes a single-input device. The bias pot has switches that allow either or - reference to be connected.

7.6.8 Readout Lines

Most analog computers have terminals on the patchpanel marked "plotter", "scope", "recorder" etc. for connection to the various readout devices. It is my feeling that this is not the best way to terminate readout devices, even on a conventional computer with a patchpanel. The display scope furnished with the 8800 and 680 computers contains a signal selector that allows any one of the 18 channels to be selected manually from the readout device itself. The advantage of this method over the conventional method of connecting output devices was brought home to me vividly the first time I used the display scope. With many different variables patched into the scope, I was able to produce any desired display by pushing a few buttons, but whenever I obtained a display worth recording permanently, I had to go back to the patchpanel and change X-Y plotter connections. The ease of selection of displays is in sharp contrast to the tedious re-patching needed to obtain hard copy output. There is no reason why a plotter or recorder shouldn't contain its own signal selector. This would not have to be a fancy arrangement with relays, logic, and lights. A rotary switch or a set of latching pushbuttons would be perfectly adequate.

This idea seems like a sound one for any analog computer, and it is especially important for a computer with automatic patching, as it allows us to treat all readout devices as interchangeable.

The proposed system assumes that every module terminates six readout lines, which have access to any of the components within the module through the module's internal matrix. The 36 readout lines are terminated on a connector on the computer console. Each channel on any readout device has a signal selector which selects one of the 36 inputs. This signal selector may be a rotary switch, or preferably, two columns of latching pushbuttons, with one column labeled 1 through 6 (for module selection, and one labeled A through F). All readout devices select from the same 36 lines.

7.7 The External and Internal Matrices

Each module has 25 component outputs, all of which have both internal and external access. Hence the external matrix has 6 X 25 = 150 matrix inputs. If each of the 62 component inputs had external access, the external matrix would have 6 X 62 = 372 matrix outputs. This 150-by-372 matrix was used as the basis for the discussions in Chapter 6, Section 6.5.

However, we have seen that component inputs, unlike component outputs, do not all need external access. Linear components, because of their input summing capability, have some internal inputs and some external ones. All non-linear components inputs have both types of access with one exception: the "S" and "C" inputs on the electronic resolver. This was because the resolver was originally conceived as a single packaged unit. After examining several problems, I decided to treat the resolver as two separate components, as described above. These extra input terminals were allowed internal access only, as a compromise between full external access and complete "burying". I now believe that it would be better to be consistent and allow external access to all non-linear component inputs, but the configuration has not been changed, to avoid invalidating programs that had already been prepared.

The "input only" components are allowed only internal access; this reduces the external matrix considerably. Counting external component inputs, we find that there are only 32 (not 62) in a module. Hence there must be 192 matrix outputs. The external matrix is thus 150 by 192, but it should be designed on the assumption that only a fraction of the inputs and outputs will be in use at one time.

Assuming 8 out of 25 component outputs per module and 16 out of 32 component inputs per module use external access, we design the external matrix as a 48-by-96 three-stage matrix. The theory of Chapter 5 dictates 10 input blocks of 5 inputs each and 12 output blocks of 8 outputs each. Expanding the input and output blocks to allow greater access,

as was done in Chapter 6, we get 15 inputs per input block (of which we expect to use only 5) and 16 outputs per output block (of which we expect to use only 8 in any one problem). The external matrix has the following structure:

10 input blocks (15 X 8)	=	1200 switches
12 output blocks (8 X 16)	=	1536 switches
8 middle blocks (10 X 12)	=	960 switches
Total switches in external matrix	=	3696 switches

All 25 component outputs have internal access, and there are 49 component inputs with internal access. Hence the internal matrix has 25 inputs and 49 outputs. Breaking the inputs into 5 blocks of 5 and the outputs into 7 blocks of 7, we have the following structure:

5 input blocks (5 X 7)	=	175 switches
7 output blocks (7 X 7)	=	343 switches
7 middle blocks (5 X 7)	=	245 switches
		763 switches

Note that a 25-by-49 rectangular matrix would require 1225 switches.

The sum of switches in both matrices is thus as follows:

One external matrix	=	3696 switches
Six internal matrices (763 switches each)	=	4578 switches
Total		8274 switches

Note that the number of internal and external switches is approximately equal, which was one of the criteria set forth in Chapter 6.

This figure does not include configuration switches (there are about 1000 of these, counting a form-C relay as two switches).

The assignment of component outputs to matrix input blocks and component inputs to matrix output blocks was done on the basis of the principles outlined in Section 6.6. Component outputs within a module are spread over as many different input blocks as possible, with special care taken to terminate interchangeable components within a module on different input blocks.

Each input block contains as many different types of components on it as possible, to facilitate interchanging of components to equalize the traffic on the various blocks. In contrast, there is deliberate group-

ing of similar components on an output block, to increase output-block fanout. The matrix assignments were punched on cards, and are printed out in tables 7-1 through 7-4. An entry such as 5-2A means module 5, component 2, input A.

7.8 The Large System

Since several of the test problems proposed by NASA were too large for a single 680, a two-console system was also defined for programming these problems. It is probably a good idea to increase the module size for larger computers, but this was not done for two reasons:

a) It allows the same internal matrix to be used in both cases, thus simplifying the programming.

b) Since we do not yet know for certain what the optimal module size is, it is probably a good idea to try the same module on two different size computers.

Hence the large matrix contains 12 modules. It has 300 matrix inputs and 384 matrix outputs. Assuming 33% input traffic and 50% output traffic , we consider it as a 100-by-192 matrix. We use 15 matrix input blocks, each with 20 matrix inputs (of which we expect to use about 7 in any problem) and 16 matrix output blocks with 24 outputs, of which we expect to use 12, and 12 middle blocks.

The switch count is as follows:

15 matrix input blocks (20 by 12)	=	3600 switches
16 matrix output blocks (12 by 24)	=	4608 switches
12 middle blocks (15 by 16)	=	2880 switches
Total switches in external matrix	=	11,088 switches
Total switches in 12 internal matrices	=	9,156 switches
Total switches for large system	=	20,224 switches

Since we have not changed the module size, but merely doubled the number of modules, the two-console system uses exactly twice as many internal switches as the single console system. The external matrix, since it grows as the 3/2 power of the number of components, should have about $2\sqrt{2} = 2.8$ times as many switches. The actual ratio is three-to-one. For both matrices, the number of internal and external switches are about equal, but the internal switches predominate in the small system and the external ones predominate in the large system.

Tables 7-5 and 7-6 give the input and output block assignments for the external matrix of the two-console system.

INPUT	INPUT BLOCK	MODULE	COMPONENT OUTPUT	INPUT	INPUT BLOCK	MODULE	COMPONENT OUTPUT	INPUT	INPUT BLOCK	MODULE	COMPONENT OUTPUT
1	1	1	22	2	1	2	23	3	1	3	24
4	1	4	25	5	1	1	15	6	1	2	16
7	1	3	17	8	1	4	18	9	1	5	19
10	1	6	20	11	1	6	21	12	1	1	12
13	1	2	13	14	1	3	14	15	1	3	11
1	2	2	22	2	2	3	23	3	2	4	24
4	2	5	25	5	2	2	15	6	2	3	16
7	2	4	17	8	2	5	18	9	2	6	19
10	2	1	20	11	2	1	21	12	2	4	12
13	2	5	13	14	2	6	14	15	2	4	11
1	3	3	22	2	3	4	23	3	3	5	24
4	3	6	25	5	3	3	15	6	3	4	16
7	3	5	17	8	3	6	18	9	3	1	19
10	3	2	20	11	3	2	21	12	3	6	13
13	3	1	14	14	3	2	12	15	3	5	11
1	4	4	22	2	4	5	23	3	4	6	24
4	4	1	25	5	4	4	15	6	4	5	16
7	4	6	17	8	4	1	18	9	4	2	19
10	4	3	20	11	4	3	21	12	4	3	13
13	4	4	14	14	4	5	12	15	4	6	11
1	5	5	22	2	5	6	23	3	5	1	24
4	5	2	25	5	5	5	15	6	5	6	16
7	5	1	17	8	5	2	18	9	5	3	19
10	5	4	20	11	5	4	21	12	5	5	14
13	5	6	12	14	5	1	13	15	5	1	11
1	6	6	22	2	6	1	23	3	6	2	24
4	6	3	25	5	6	6	15	6	6	1	16
7	6	2	17	8	6	3	18	9	6	4	19
10	6	5	20	11	6	5	21	12	6	2	14
13	6	3	12	14	6	4	13	15	6	2	11
1	7	1	1	2	7	2	2	3	7	3	3
4	7	4	4	5	7	5	5	6	7	6	6
7	7	5	1	8	7	6	2	9	7	1	3
10	7	1	7	11	7	2	8	12	7	3	9
13	7	4	10	14	7	5	7	15	7	6	8
1	8	2	1	2	8	3	2	3	8	4	3
4	8	5	4	5	8	6	5	6	8	1	6
7	8	2	4	8	8	3	5	9	8	4	6
10	8	1	8	11	8	2	9	12	8	3	10
13	8	6	7	14	8	5	8	15	8	4	9
1	9	3	1	2	9	4	2	3	9	5	3
4	9	6	4	5	9	1	5	6	9	2	6
7	9	6	1	8	9	1	2	9	9	2	3
10	9	5	9	11	9	1	10	12	9	4	7
13	9	3	8	14	9	6	9	15	9	2	10
1	10	4	1	2	10	5	2	3	10	6	3
4	10	1	4	5	10	2	5	6	10	3	6
7	10	3	4	8	10	4	5	9	10	5	6
10	10	5	10	11	10	2	7	12	10	4	8
13	10	1	9	14	10	6	10	15	10	3	7

Table 7-1. Terminal Connections for External Matrix Input Blocks (Single-Console System).

OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT	INPUT
1	1	1	12	
1	4	2	12	
1	7	3	12	
1	10	1	7	A
1	13	1	15	N
1	16	1	22	
2	1	4	12	
2	4	5	12	
2	7	6	12	
2	10	1	8	A
2	13	1	15	D
2	16	1	23	
3	1	1	16	N
3	4	1	19	N
3	7	2	18	N
3	10	3	17	N
3	13	2	15	N
3	16	2	22	
4	1	1	16	D
4	4	1	19	D
4	7	2	18	D
4	10	3	17	D
4	13	2	15	D
4	16	2	23	
5	1	4	16	N
5	4	4	19	N
5	7	5	18	N
5	10	6	17	N
5	13	3	15	N
5	16	3	22	
6	1	4	16	D
6	4	4	19	D
6	7	5	18	D
6	10	6	17	D
6	13	3	15	D
6	16	3	23	

OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT	INPUT
1	2	1	13	
1	5	2	13	
1	8	3	13	
1	11	2	7	A
1	14	1	20	N
1	16	1	22	
2	2	4	13	
2	5	5	13	
2	8	6	13	
2	11	2	8	A
2	14	1	21	D
2	16	1	23	
3	2	1	17	N
3	5	2	16	N
3	8	2	19	N
3	11	3	18	N
3	14	2	20	N
3	16	2	22	
4	2	1	17	D
4	5	2	16	D
4	8	2	19	D
4	11	3	18	D
4	14	2	21	D
4	16	2	23	
5	2	4	17	N
5	5	5	16	N
5	8	5	19	N
5	11	6	18	N
5	14	3	20	N
5	16	3	22	
6	2	4	17	D
6	5	5	16	D
6	8	5	19	D
6	11	6	18	D
6	14	3	21	D
6	16	3	23	

OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT	INPUT
1	3	1	14	
1	6	2	14	
1	9	3	14	
1	12	3	7	A
1	15	1	24	X
1	16	1	22	
2	3	4	14	
2	6	5	14	
2	9	6	14	
2	12	3	8	A
2	15	1	25	Y
2	16	1	23	
3	3	1	18	N
3	6	2	17	N
3	9	3	16	N
3	12	3	19	N
3	15	2	24	N
3	16	2	22	
4	3	1	18	D
4	6	2	17	D
4	9	3	16	D
4	12	3	14	D
4	15	2	25	Y
4	16	2	23	
5	3	4	18	N
5	6	5	17	N
5	9	6	16	N
5	12	6	19	N
5	15	3	24	X
5	16	3	22	
6	3	4	18	D
6	6	5	17	D
6	9	6	16	D
6	12	6	19	D
6	15	3	25	Y
6	16	3	23	

Table 7-2. Terminal Connections for External Matrix Output Blocks (Single - Console System). First of two sheets.

OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT	INPUT	OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT	INPUT	OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT	INPUT
7	1	1	9	A	7	2	1	10	A	7	3	1	11	A
7	4	2	9	A	7	5	2	10	A	7	6	2	11	A
7	7	3	9	A	7	8	3	10	A	7	9	3	11	A
7	10	4	7	A	7	11	5	7	A	7	12	6	7	A
7	13	4	15	N	7	14	4	20	N	7	15	4	24	X
7	16	4	22		7	16	4	22		7	16	4	22	
8	1	4	9	A	8	2	4	10	A	8	3	4	11	A
8	4	5	9	A	8	5	5	10	A	8	6	5	11	A
8	7	6	9	A	8	8	6	10	A	8	9	6	11	A
8	10	4	8	A	8	11	5	8	A	8	12	6	8	A
8	13	4	15	D	8	14	4	21	D	8	15	4	25	Y
8	16	4	23		8	16	4	23		8	16	4	23	
9	1	1	1	A	9	2	1	2	A	9	3	2	1	A
9	4	2	2	A	9	5	3	1	A	9	6	3	2	A
9	7	4	1	A	9	8	4	2	A	9	9	5	1	A
9	10	5	2	A	9	11	6	1	A	9	12	6	2	A
9	13	5	15	N	9	14	5	20	N	9	15	5	24	X
9	16	5	22		9	16	5	22		9	16	5	22	
10	1	1	1	B	10	2	1	2	B	10	3	2	1	B
10	4	2	2	B	10	5	3	1	B	10	6	3	2	B
10	7	4	1	B	10	8	4	2	B	10	9	5	1	B
10	10	5	2	B	10	11	6	1	B	10	12	6	2	B
10	13	5	15	D	10	14	5	21	D	10	15	5	25	Y
10	16	5	23		10	16	5	23		10	16	5	23	
11	1	1	3	A	11	2	1	4	A	11	3	2	3	A
11	4	2	4	A	11	5	3	3	A	11	6	3	4	A
11	7	4	3	A	11	8	4	4	A	11	9	5	3	A
11	10	5	4	A	11	11	6	3	A	11	12	6	4	A
11	13	6	15	N	11	14	6	20	N	11	15	6	24	X
11	16	6	22		11	16	6	22		11	16	6	22	
12	1	1	5	A	12	2	1	6	A	12	3	2	5	A
12	4	2	6	A	12	5	3	5	A	12	6	3	6	A
12	7	4	5	A	12	8	4	6	A	12	9	5	5	A
12	10	5	6	A	12	11	6	5	A	12	12	6	6	A
12	13	6	15	D	12	14	6	21	D	12	15	6	25	Y
12	16	6	23		12	16	6	23		12	16	6	23	

Table 7-2 (Continued).

INPUT	INPUT BLOCK	MODULE	COMPONENT OUTPUT
1	1	1	1
4	1	1	5
2	2	1	7
5	2	1	22
3	3	1	13
1	4	1	4
4	4	1	18
2	5	1	10
5	5	1	25

INPUT	INPUT BLOCK	MODULE	COMPONENT OUTPUT
2	1	1	6
5	1	1	21
3	2	1	12
1	3	1	3
4	3	1	17
2	4	1	9
5	4	1	24
3	5	1	20
5	5	1	25

INPUT	INPUT BLOCK	MODULE	COMPONENT OUTPUT
3	1	1	11
1	2	1	2
4	2	1	16
2	3	1	8
5	3	1	23
3	4	1	14
1	5	1	5
4	5	1	19
5	5	1	25

Table 7-3. Terminal Connections for Internal Matrix Input Blocks.

OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT INPUT	OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT INPUT	OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT INPUT
1	1	1	15 A	1	2	1	16 A	1	3	1	17 A
1	4	1	18 A	1	X5	1	19 A	1	6	1	24
1	7	1	29 A	2	1	1	15 B	2	2	1	16 B
2	3	1	17 B	2	4	1	18 B	2	5	1	19 B
2	6	1	24 S	2	7	1	29 B	3	1	1	1 C
3	2	1	2 C	3	3	1	3 B	3	4	1	5 B
3	5	1	6 B	3	6	1	26	3	7	1	29 C
4	1	1	1 D	4	2	1	2 D	4	3	1	5 C
4	4	1	6 C	4	5	1	7 B	4	6	1	8 B
4	7	1	29 D	5	1	1	5 D	5	2	1	6 D
5	3	1	7 C	5	4	1	8 C	5	5	1	9 B
5	6	1	10 B	5	7	1	27	6	1	1	12 F
6	2	1	13 F	6	3	1	14 F	6	4	1	20 A
6	5	1	22 F	6	6	1	25	6	7	1	29 E
7	1	1	4 B	7	2	1	11 B	7	3	1	21 B
7	4	1	23 F	7	5	1	25 C	7	6	1	28
7	7	1	29 F	7	7	1	29 F	7	7	1	29 F

Table 7-4. Terminal Connections for Internal Matrix Output Blocks.

INPUT	INPUT BLOCK	MODULE	COMPONENT OUTPUT
1	1	1	1
2	1	1	16
3	1	2	6
4	1	2	21
5	1	3	11
6	1	4	12
7	1	4	2
8	1	5	17
9	1	5	7
10	1	6	22
11	1	7	8
12	1	7	23
13	1	8	13
14	1	8	3
15	1	9	18
16	1	10	19
17	1	10	9
18	1	11	24
19	1	11	14
20	1	12	4
1	4	1	4
2	4	1	19
3	4	2	9
4	4	2	24
5	4	3	14
6	4	4	15
7	4	4	5
8	4	5	20
9	4	5	10
10	4	6	25
11	4	7	11
12	4	7	1
13	4	8	16
14	4	8	6
15	4	9	21
16	4	10	22
17	4	10	12
18	4	11	2
19	4	11	17
20	4	12	7
1	7	1	7
2	7	1	22
3	7	2	12
4	7	3	2
5	7	3	17
6	7	4	18
7	7	4	8
8	7	5	23
9	7	6	13
10	7	6	3
11	7	7	14
12	7	7	4
13	7	8	19
14	7	9	9

INPUT	INPUT BLOCK	MODULE	COMPONENT OUTPUT
1	2	1	2
2	2	1	17
3	2	2	7
4	2	2	22
5	2	3	12
6	2	4	13
7	2	4	3
8	2	5	18
9	2	5	8
10	2	6	23
11	2	7	9
12	2	7	24
13	2	8	14
14	2	8	4
15	2	9	19
16	2	10	20
17	2	10	10
18	2	11	25
19	2	11	15
20	2	12	5
1	5	1	5
2	5	1	20
3	5	2	10
4	5	2	25
5	5	3	15
6	5	4	16
7	5	4	6
8	5	5	21
9	5	5	11
10	5	6	1
11	5	7	12
12	5	7	2
13	5	8	17
14	5	8	7
15	5	9	22
16	5	10	23
17	5	10	13
18	5	11	3
19	5	11	18
20	5	12	8
1	8	1	8
2	8	1	23
3	8	2	13
4	8	3	3
5	8	3	18
6	8	4	19
7	8	4	9
8	8	5	24
9	8	6	14
10	8	6	4
11	8	7	15
12	8	7	5
13	8	8	20
14	8	9	10

INPUT	INPUT BLOCK	MODULE	COMPONENT OUTPUT
1	3	1	3
2	3	1	18
3	3	2	8
4	3	2	23
5	3	3	13
6	3	4	14
7	3	4	4
8	3	5	19
9	3	5	9
10	3	6	24
11	3	7	10
12	3	7	25
13	3	8	15
14	3	8	5
15	3	9	20
16	3	10	21
17	3	10	11
18	3	11	1
19	3	11	16
20	3	12	6
1	6	1	6
2	6	1	21
3	6	2	11
4	6	3	1
5	6	3	16
6	6	4	17
7	6	4	7
8	6	5	22
9	6	6	12
10	6	6	2
11	6	7	13
12	6	7	3
13	6	8	18
14	6	9	8
15	6	9	23
16	6	10	24
17	6	10	14
18	6	11	4
19	6	12	19
20	6	12	9
1	9	1	9
2	9	1	24
3	9	2	14
4	9	3	4
5	9	3	19
6	9	4	20
7	9	4	10
8	9	5	25
9	9	6	15
10	9	6	5
11	9	7	16
12	9	7	6
13	9	8	21
14	9	9	11

Table 7-5. Terminal Connections for External Matrix Input Blocks (Two-Console System). First of two sheets.

INPUT	INPUT BLOCK	MODULE	COMPONENT OUTPUT	INPUT	INPUT BLOCK	MODULE	COMPONENT OUTPUT	INPUT	INPUT BLOCK	MODULE	COMPONENT OUTPUT
15	7	9	24	15	8	9	25	15	9	9	1
16	7	10	25	16	8	10	1	16	9	10	2
17	7	10	15	17	8	10	16	17	9	10	17
18	7	11	5	18	8	11	6	18	9	11	7
19	7	12	20	19	8	12	21	19	9	12	22
20	7	12	10	20	8	12	11	20	9	12	12
1	10	1	10	1	11	1	11	1	12	1	12
2	10	1	25	2	11	2	1	2	12	2	2
3	10	2	15	3	11	2	16	3	12	2	17
4	10	3	5	4	11	3	6	4	12	3	7
5	10	3	20	5	11	3	21	5	12	3	22
6	10	4	21	6	11	4	22	6	12	4	23
7	10	4	11	7	11	5	12	7	12	5	13
8	10	5	1	8	11	5	2	8	12	5	3
9	10	6	16	9	11	6	17	9	12	6	18
10	10	6	6	10	11	6	7	10	12	6	8
11	10	7	17	11	11	7	18	11	12	7	19
12	10	7	7	12	11	8	8	12	12	8	9
13	10	8	22	13	11	8	23	13	12	8	24
14	10	9	12	14	11	9	13	14	12	9	14
15	10	9	2	15	11	9	3	15	12	9	4
16	10	10	3	16	11	10	4	16	12	10	5
17	10	10	18	17	11	11	19	17	12	11	20
18	10	11	8	18	11	11	9	18	12	11	10
19	10	12	23	19	11	12	24	19	12	12	25
20	10	12	13	20	11	12	14	20	12	12	15
1	13	1	13	1	14	1	14	1	15	1	15
2	13	2	3	2	14	2	4	2	15	2	5
3	13	2	18	3	14	2	19	3	15	2	20
4	13	3	8	4	14	3	9	4	15	3	10
5	13	3	23	5	14	3	24	5	15	3	25
6	13	4	24	6	14	4	25	6	15	4	1
7	13	5	14	7	14	5	15	7	15	5	16
8	13	5	4	8	14	5	5	8	15	5	6
9	13	6	19	9	14	6	20	9	15	6	21
10	13	6	9	10	14	6	10	10	15	6	11
11	13	7	20	11	14	7	21	11	15	7	22
12	13	8	10	12	14	8	11	12	15	8	12
13	13	8	25	13	14	8	1	13	15	8	2
14	13	9	15	14	14	9	16	14	15	9	17
15	13	9	5	15	14	9	6	15	15	9	7
16	13	10	6	16	14	10	7	16	15	10	8
17	13	11	21	17	14	11	22	17	15	11	23
18	13	11	11	18	14	11	12	18	15	11	13
19	13	12	1	19	14	12	2	19	15	12	3
20	13	12	16	20	14	12	17	20	15	12	18

Table 7-5. (Continued)

OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT INPUT	OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT INPUT	OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT INPUT
1	1	1	12	2	1	4	12	3	1	7	12
1	2	1	13	2	2	4	13	3	2	7	13
1	3	1	14	2	3	4	14	3	3	7	14
1	4	2	12	2	4	5	12	3	4	8	12
1	5	2	13	2	5	5	13	3	5	8	13
1	6	2	14	2	6	5	14	3	6	8	14
1	7	3	12	2	7	6	12	3	7	9	12
1	8	3	13	2	8	6	13	3	8	9	13
1	9	3	14	2	9	6	14	3	9	9	14
1	10	1	11 A	2	10	4	11 A	3	10	7	11 A
1	11	2	11 A	2	11	5	11 A	3	11	8	11 A
1	12	3	11 A	2	12	6	11 A	3	12	9	11 A
1	13	1	19 N	2	13	1	19 D	3	13	7	19 N
1	14	2	19 N	2	14	2	19 D	3	14	8	19 N
1	15	3	19 N	2	15	3	19 D	3	15	9	19 N
1	16	4	19 N	2	16	4	19 D	3	16	10	19 N
1	17	5	19 N	2	17	5	19 D	3	17	11	19 N
1	18	6	19 N	2	18	6	19 D	3	18	12	19 N
1	19	1	20 N	2	19	1	21 D	3	19	2	20 N
1	20	7	20 N	2	20	7	21 D	3	20	8	20 N
1	21	1	22	2	21	1	23	3	21	2	22
1	22	7	22	2	22	7	23	3	22	8	22
1	23	1	24 X	2	23	1	25 Y	3	23	2	24 X
1	24	7	24 X	2	24	7	25 Y	3	24	8	24 X
4	1	10	12	5	1	1	15 N	6	1	1	15 D
4	2	10	13	5	2	1	16 N	6	2	1	16 D
4	3	10	14	5	3	2	15 N	6	3	2	15 D
4	4	11	12	5	4	2	16 N	6	4	2	16 D
4	5	11	13	5	5	3	15 N	6	5	3	15 D
4	6	11	14	5	6	3	16 N	6	6	3	16 D
4	7	12	12	5	7	4	15 N	6	7	4	15 D
4	8	12	13	5	8	4	16 N	6	8	4	16 D
4	9	12	14	5	9	5	15 N	6	9	5	15 D
4	10	10	11 A	5	10	5	16 N	6	10	5	16 D
4	11	11	11 A	5	11	6	15 N	6	11	6	15 D
4	12	12	11 A	5	12	6	16 N	6	12	6	16 D
4	13	7	19 D	5	13	1	7 A	6	13	4	7 A
4	14	8	19 D	5	14	1	8 A	6	14	4	8 A
4	15	9	19 D	5	15	2	7 A	6	15	5	7 A
4	16	10	19 D	5	16	2	8 A	6	16	5	8 A
4	17	11	19 D	5	17	3	7 A	6	17	6	7 A
4	18	12	19 D	5	18	3	8 A	6	18	6	8 A
4	19	2	21 D	5	19	3	20 N	6	19	3	21 D
4	20	8	21 D	5	20	9	20 N	6	20	9	21 D
4	21	2	23	5	21	3	22	6	21	3	23
4	22	8	23	5	22	9	22	6	22	9	23
4	23	2	25 Y	5	23	3	24 X	6	23	3	25 Y
4	24	8	25 Y	5	24	9	24 X	6	24	9	25 Y
7	1	7	15 N	8	1	7	15 D	9	1	1	17 N
7	2	7	16 N	8	2	7	16 D	9	2	1	18 N
7	3	8	15 N	8	3	8	15 D	9	3	2	17 N
7	4	8	16 N	8	4	8	16 D	9	4	2	18 N
7	5	9	15 N	8	5	9	15 D	9	5	3	17 N
7	6	9	16 N	8	6	9	16 D	9	6	3	18 N

Table 7-6. Terminal Connections for External Matrix Output Blocks (Two-console System). First of three sheets.

OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT INPUT		OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT INPUT		OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT INPUT	
7	7	10	15	N	8	7	10	15	D	9	7	4	17	N
7	8	10	16	N	8	8	10	16	D	9	8	4	18	N
7	9	11	15	N	8	9	11	15	D	9	9	5	17	N
7	10	11	16	N	8	10	11	16	D	9	10	5	18	N
7	11	12	15	N	8	11	12	15	D	9	11	6	17	N
7	12	12	16	N	8	12	12	16	D	9	12	6	18	N
7	13	7	7	A	8	13	10	7	A	9	13	1	9	A
7	14	7	8	A	8	14	10	8	A	9	14	1	10	A
7	15	8	7	A	8	15	11	7	A	9	15	2	9	A
7	16	8	8	A	8	16	11	8	A	9	16	2	10	A
7	17	9	7	A	8	17	12	7	A	9	17	3	9	A
7	18	9	8	A	8	18	12	8	A	9	18	3	10	A
7	19	4	20	N	8	19	4	21	D	9	19	5	20	N
7	20	10	20	N	8	20	10	21	D	9	20	11	20	N
7	21	4	22		8	21	4	23		9	21	5	22	
7	22	10	22		8	22	10	23		9	22	11	22	
7	23	4	24	X	8	23	4	25	Y	9	23	5	24	X
7	24	10	24	X	8	24	10	25	Y	9	24	11	24	X
10	1	1	17	D	11	1	7	17	N	12	1	7	17	D
10	2	1	18	D	11	2	7	18	N	12	2	7	18	D
10	3	2	17	D	11	3	8	17	N	12	3	8	17	D
10	4	2	18	D	11	4	8	18	N	12	4	8	18	D
10	5	3	17	D	11	5	9	17	N	12	5	9	17	D
10	6	3	18	D	11	6	9	18	N	12	6	9	18	D
10	7	4	17	D	11	7	10	17	N	12	7	10	17	D
10	8	4	18	D	11	8	10	18	N	12	8	10	18	D
10	9	5	17	D	11	9	11	17	N	12	9	11	17	D
10	10	5	18	D	11	10	11	18	N	12	10	11	18	D
10	11	6	17	D	11	11	12	17	N	12	11	12	17	D
10	12	6	18	D	11	12	12	18	N	12	12	12	18	D
10	13	4	9	A	11	13	7	9	A	12	13	10	9	A
10	14	4	10	A	11	14	7	10	A	12	14	10	10	A
10	15	5	9	A	11	15	8	9	A	12	15	11	9	A
10	16	5	10	A	11	16	8	10	A	12	16	11	10	A
10	17	6	9	A	11	17	9	9	A	12	17	12	9	A
10	18	6	10	A	11	18	9	10	A	12	18	12	10	A
10	19	5	21	D	11	19	6	20	N	12	19	6	21	D
10	20	11	21	D	11	20	12	20	N	12	20	12	21	D
10	21	5	23		11	21	6	22		12	21	6	23	
10	22	11	23		11	22	12	22		12	22	12	23	
10	23	5	25	Y	11	23	6	24	X	12	23	6	25	Y
10	24	11	25	Y	11	24	12	24	X	12	24	12	25	Y
13	1	1	1	A	14	1	7	1	A	15	1	1	1	B
13	2	1	2	A	14	2	7	2	A	15	2	1	2	B
13	3	1	3	A	14	3	7	3	A	15	3	1	5	A
13	4	1	4	A	14	4	7	4	A	15	4	1	6	A
13	5	2	1	A	14	5	8	1	A	15	5	2	1	B
13	6	2	2	A	14	6	8	2	A	15	6	2	2	B
13	7	2	3	A	14	7	8	3	A	15	7	2	5	A
13	8	2	4	A	14	8	8	4	A	15	8	2	6	A
13	9	3	1	A	14	9	9	1	A	15	9	3	1	B
13	10	3	2	A	14	10	9	2	A	15	10	3	2	B
13	11	3	3	A	14	11	9	3	A	15	11	3	5	A
13	12	3	4	A	14	12	9	4	A	15	12	3	6	A

Table 7-6 (Continued).

OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT INPUT	OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT INPUT	OUTPUT BLOCK	OUTPUT	MODULE	COMPONENT INPUT
13	13	4	1 A	14	13	10	1 A	15	13	4	1 B
13	14	4	2 A	14	14	10	2 A	15	14	4	2 B
13	15	4	3 A	14	15	10	3 A	15	15	4	5 A
13	16	4	4 A	14	16	10	4 A	15	16	4	6 A
13	17	5	1 A	14	17	11	1 A	15	17	5	1 B
13	18	5	2 A	14	18	11	2 A	15	18	5	2 B
13	19	5	3 A	14	19	11	3 A	15	19	5	5 A
13	20	5	4 A	14	20	11	4 A	15	20	5	6 A
13	21	6	1 A	14	21	12	1 A	15	21	6	1 B
13	22	6	2 A	14	22	12	2 A	15	22	6	2 B
13	23	6	3 A	14	23	12	3 A	15	23	6	5 A
13	24	6	4 A	14	24	12	4 A	15	24	6	6 A
16	1	7	1 B	16	2	7	2 B	16	3	7	5 A
16	4	7	6 A	16	5	8	1 B	16	6	8	2 B
16	7	8	5 A	16	8	8	6 A	16	9	9	1 B
16	10	9	2 B	16	11	9	5 A	16	12	9	6 A
16	13	10	1 B	16	14	10	2 B	16	15	10	5 A
16	16	10	6 A	16	17	11	1 B	16	18	11	2 B
16	19	11	5 A	16	20	11	6 A	16	21	12	1 B
16	22	12	2 B	16	23	12	5 A	16	24	12	6 A

Table 7-6 (Continued).

8. RESULTS OF NASA SAMPLE PROGRAMS

The adequacy of the proposed systems can only be demonstrated by programming actual problems. NASA has submitted five problems for analysis. One of these is in two parts which are sufficiently different that they can be considered as two different problems: phase 1 and phase 3 of the Saturn IV Stage Control Study.

All but one of these problems have been implemented on the proposed systems. The fifth (the J-2 rocket engine) appears on visual inspection to be sufficiently straightforward that it could be programmed fairly easily using eight modules, with fairly light traffic on the external matrix, but I did not actually go through the details. It is quite clear from the problems already programmed that the traffic is quite a bit lighter than originally anticipated. In fact, the systems are not merely adequate, but probably overdesigned by about 10 or 20%. None of the problems took more than 6 of the 8 middle blocks on the single-console system or 10 of the 12 on the two-console system; and these results were achieved without any interchanging of components to equalize input and output blocks.

Two of the analog programs are presented in detail in this chapter, complete with analog diagrams, connection statements, and computer printout of the programming arrays. These should be enough to give the reader a "feel" for programming the recommended systems. Programs for the other problems, although not included in this report, will be furnished to the contractor upon request.

8.1 Voyager, First Stage Ascent

This program uses essentially the same equations as the NASA version (Reference 5) with the following modifications. (References to page numbers are to the NASA publication; references to equation numbers are to the listing given later in this section).

(a) The last equation on page 5 ($\sin \alpha = u/V$) requires a division which becomes O/O when V = O. However, the variable $\sin \alpha$ is used only in two places, and in both equations it is multiplied by V^2. Hence the division is not really necessary. The singularity may be removed by substitution and cancellation. The same thing holds true for the q/V expression in the $\dot{q}$ equation on page 10. The NASA program used feedback limiters and relays to "pacify" these division circuits when V = O, but this is not necessary if the singularities are cancelled out.

(b) Certain complicated algebraic expressions were given explicit names to facilitate labelling the outputs on the diagram. New equations defining these variables were introduced. (equations 8A, 8B, 9A, 9B, and 10A).

(c) On page 7, the sloshing accelerations ($\ddot{\lambda}_i$) are defined in terms of the variables FL_i. Further down the page, the FL_i are defined in terms of the $\ddot{\lambda}_i$. This leads to algebraic loops which can be easily eliminated by algebraic substitution. Furthermore, the sloshing mass ms_i may be entirely removed from the loop. This leads to equations 27 through 28.

(d) Redundant integration in the equation for M_t may be eliminated by writing the equations in terms of M_t. This leads to equations 16 through 20.

Equations - Voyager First Stage

1) $\beta_c = a_0(\phi - \phi_c) + a_1 \dot{q}$

2) $F_1 = a_2(\beta_c - \beta_1) - a_3 \dot{\beta}_1'$

3) $F_2 = a_4(\beta_c - \beta_2) - a_5 \dot{\beta}_2'$

4) $\beta_1 = \beta_1' + \Delta\beta$

5) $\beta_2 = \beta_2' + \Delta\beta$

6) $\ddot{\beta}_1' = (R_1/L_1)F_1 - a_6 \dot{\beta}_1'$

7) $\ddot{\beta}_2' = (R_2/L_2)F_2 - a_7 \dot{\beta}_2'$

8) $\dot{v}' = A_8 - G\cos\phi + u'q$

8A) $A_8 = 1/M_t(B_8 + T_1 + T_2 + T_3 - \Sigma FN_i)$

8B) $B_8 = -\frac{1}{2} C D_o \rho V^2 S$

9) $\dot{u}' = A_9 + G\sin\phi - v'q$

9A) $A_9 = 1/M_t(B_9 + T_1(\beta_1 + \beta_2) + \Sigma FL_i)$

9B) $B_9 = -\frac{1}{2} C_{m\alpha} u \rho V S$

10) $\dot{q} = 1/I_{zz}(A_{10} - L T_1(\beta_1 + \beta_2) + \Sigma FN_i \lambda_i + \Sigma FL_i b_i)$

10A) $A_{10} = \frac{d}{2}(C_{m\alpha} u + C_{mq} q)\rho V S$

11) $\phi = \phi' + \Delta\phi$

12) $\dot{\phi}' = q$

13) $\dot{v} = v\cos\phi - u\sin\phi$

14) $\dot{x} = v\sin\phi + u\cos\phi$

15) $C_{m\alpha}$, $C_{n\alpha}$, C_{D_0}, C_{mq}, $1/I_{zz}$, L, and P are functions of M.

16) $\dot{M}_t = 1/GI_{sp}\left(T_1+T_2+T_3\right)$

17) $T = T_o + A_o(P_o - P)$

18) $T_1 = T/4$ or 0 (switched)

19) $T_2 = T_1$

20) $T_3 = T/2$ or 0 (switched)

21) $M = V/V_s$

22) $V = \sqrt{v^2+u^2}$

23) $\ell_1 = \ell_2 = \ell = d/4$ (constant)

24) $u = u' + V_w\cos\phi$

25) $v = v' + V_w\sin\phi$

26) V_w, $1/V_s$, ρ, and P are functions of Y

27) $\ddot{\lambda}_i = \frac{1}{2}\left[g\sin\phi - \dot{u}' - b_i\dot{q} + q^2\lambda_i - \omega_i^2\lambda_i\right]$ $\quad i = 1, 2, 3$

28) $FL_i = ms_i\left(\ddot{\lambda}_i + \omega_i^2\lambda_i\right)$ $\quad i = 1, 2, 3$

29) $FN_i = ms_i\left[g\cos\phi + \dot{v}' - b_i q^2 - 2\dot{\lambda}_i q - \lambda_i\dot{q}\right]$ $\quad i = 1, 2, 3$

30) $\Delta\phi$, ΔCG, ϕ_c, ms_i, ω_i^2, and b_i $(i = 1, 2, 3)$ are functions of time.

31) $\Delta B = \Delta CG/L$

Figure 8-1 shows the analog diagram. Table 8-1 shows the listing of connections.

A single line in this table provides all the interconnection statements for a single component. The first numbers entered identify the module and the component within the module. The subsequent numbers identify the various inputs of this component and define what they are connected to. For example, the first line says that component 1-3 (the third component in module number one) receives its "A" input from module 3, component 15, and its "B" input from module 1, component 15.

Each of these lines corresponds to a single punched card, so that the entire program is defined by one card per component. In an operating system, symbols for the component's operational mode would also be included (whether it is a summer or an integrator, whether it is a multiplier or divider, etc). These would operate the configuration relays within the component.

Table 8-2 gives the printout of the programming array. Below the matrix is a printout of the following variables:

IS; the number of inputs per input block (called n in Chapter 4)

JS; the number of input blocks (called X in Chapter 4)

KS; the number of output blocks (called Z in Chapter 4)

LS; a matrix identifier. LS = 0 identifies this as the main (external) matrix, LS = i identifies the internal matrix for module i.

MSMAX; the maximum number of middle blocks allowed. If the algorithm uses more than this many middle blocks, an error message will be generated. In an operating system, LSMAX would be set equal to the number of middle blocks actually present in the hardware; for this study, it was taken to be large enough to allow virtually all programs to be implemented.

ALG; the identifier for the algorithm used. Several algorithms were tried, the one described in Chapter 9 and used in this report is number 3.

MS; the number of middle blocks actually required by this particular program.

Below this line are printed the middle blocks assigned for each input block.

Table 8 - 3 gives the same information for each of the six internal matrices.

Note that only 10 (not 12) middle blocks are used in the external matrix, and a maximum of 5 (not 7) middle blocks are used in the internal matrices.

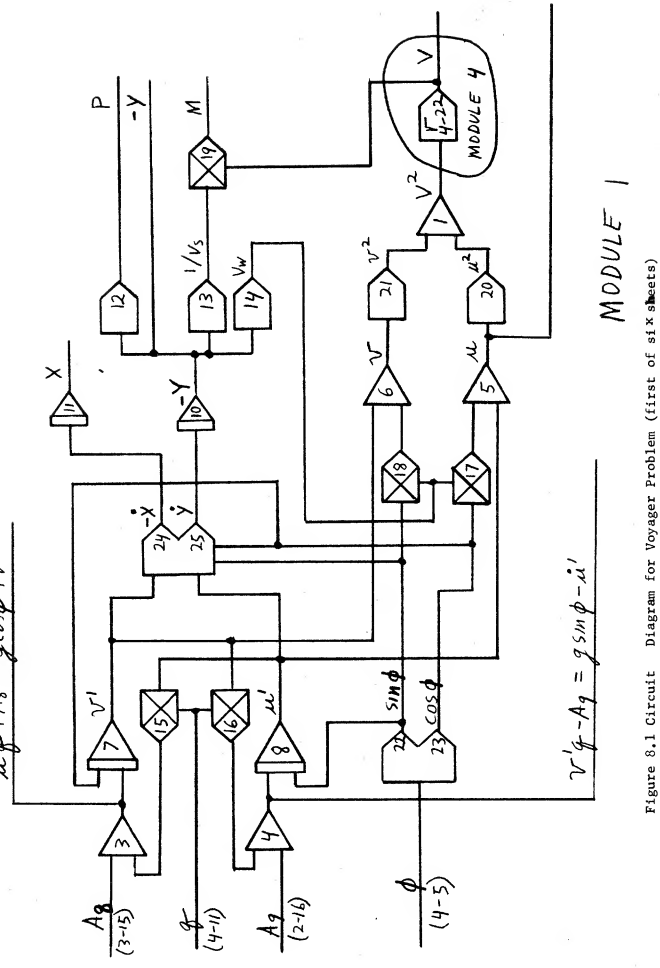
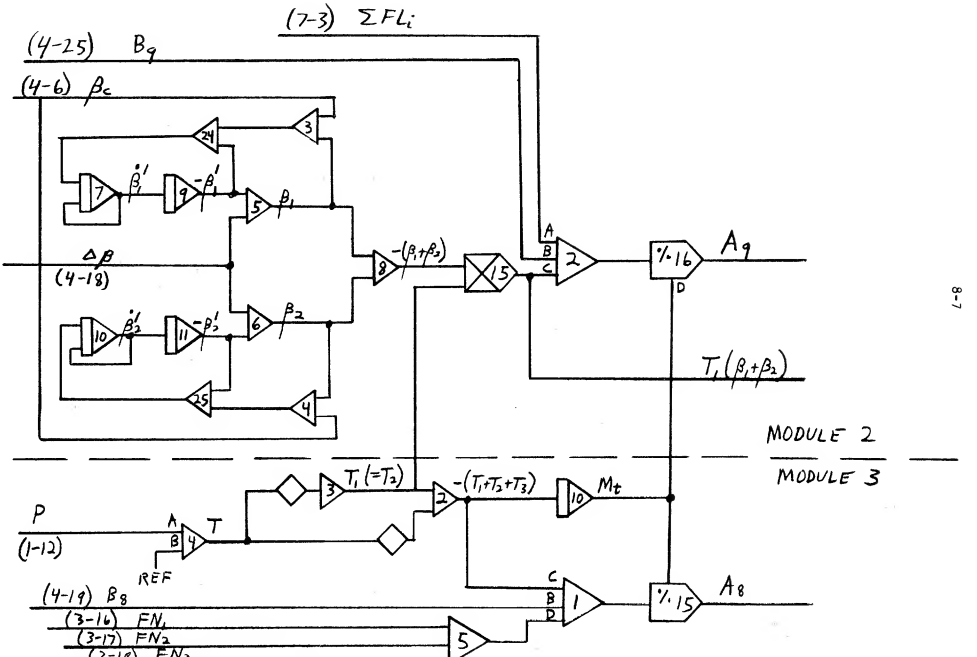
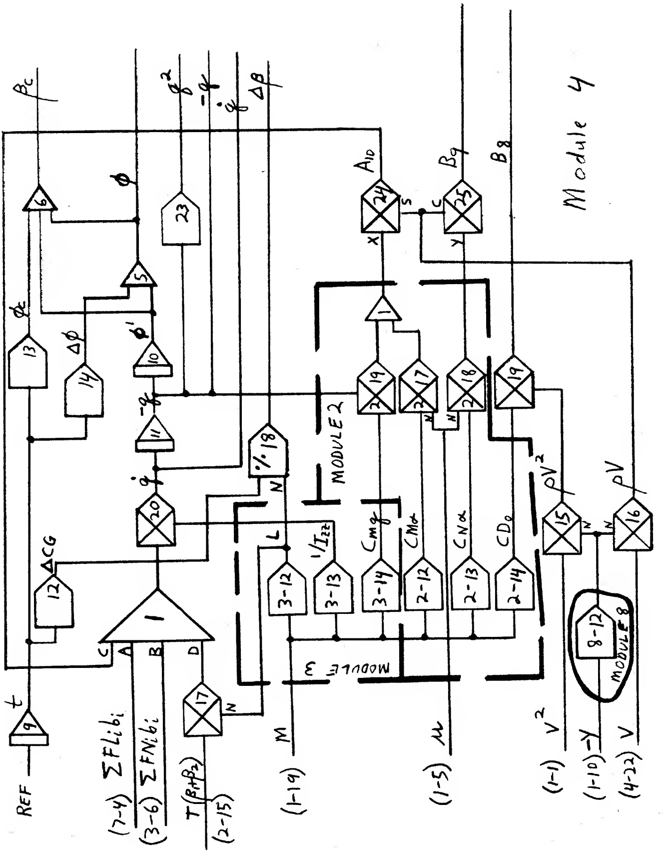


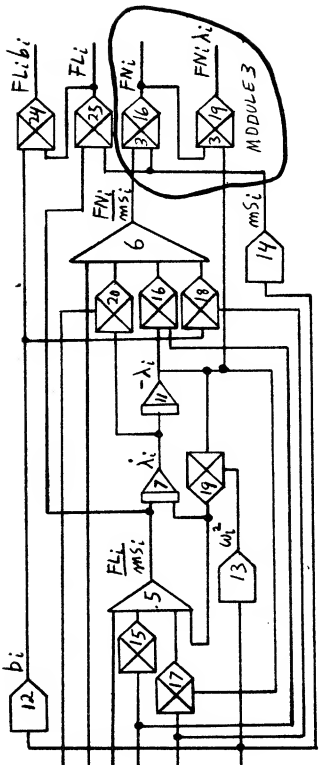
Figure 8.1 Circuit Diagram for Voyager Problem (first of six sheets)


(7-3) ΣFLi
(4-25) Bq
(4-6) βc
24
3
7
β'1
9
-β'1
5
β1
Δβ
(4-18)
8
-(β1+β2)
15
A
B
C
2
1/16
Aq
D
10
β'2
11
-β'2
6
β2
25
4
T1(β1+β2)
MODULE 2
MODULE 3
3
T1 (=T2)
-(T1+T2+T3)
10
Mt
P
(1-12)
A
B
4
T
REF
C
B
D
1
1/15
A8
(4-19) B8
(3-16) FN1
(3-17) FN2
(3-18) FN3
5

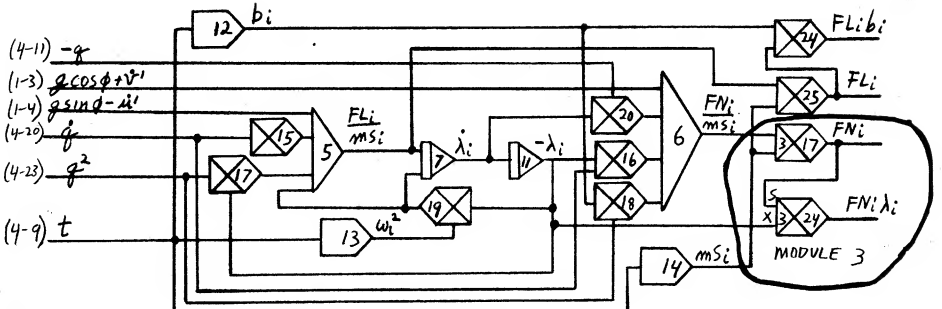


Module 4
MODULE 2
MODULE 3
MODULE 8
REF
(7-4) ΣFLibi
(3-6) ΣFNibi
T (B1+B2) (2-15)
(1-19) M
(1-5) μ
(1-1) V²
(1-10) -Y
(4-22) V
ΔCG
ρV²
ρV
CD0
CNα
CMα
Cmq
1/Izz
Bc
Bq
B8
A10
ΔB

MODULE 5 (i=1)


(4-11) $\ddot{x}$
(1-3) $g\cos\phi\cos\lambda'$
(1-4) $g\sin\phi - \lambda'^2$
(4-20) $\ddot{y}$
(4-23) $\ddot{z}$
(4-9) t
b_i
12
FL_i/b_i
24
FL_i
25
FW_i
16
FW_i/λ_i
19
MODULE 3
FW_i ms$_i$
6
20
16
18
14
ms$_i$
$-\lambda_i$
11
λ_i
7
19
ω_i^2
13
FL_i ms$_i$
5
15
17

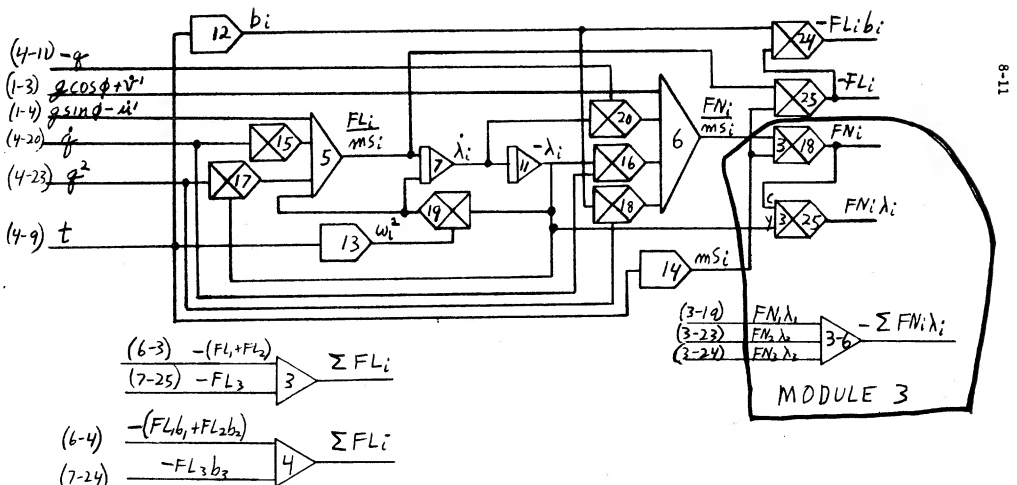
MODULE 6 ($i = 2$)



(5-25) FL_1
(6-25) FL_2
3 $-(FL_1 + FL_2)$

(5-24) $FL_1 b_1$
(6-24) $FL_2 b_2$
4 $-(FL_1 b_1 + FL_2 b_2)$

MODULE 7 ($i=3$)


12 b_i
(4-11) $-\ddot{x}$
(1-3) $g\cos\phi+\ddot{v}'$
(1-4) $g\sin\phi-\ddot{u}'$
(4-20) $\dot{q}$
(4-23) q^2
(4-9) t
15
17
5
FL_i / ms_i
7 $\dot{\lambda}_i$
11 $-\lambda_i$
13 ω_i^2
19
20
16
18
6
FN_i / ms_i
14 ms_i
24 $-FL_i b_i$
25 $-FL_i$
3 18 FN_i
3 25 $FN_i\lambda_i$
(3-19) $FN_1\lambda_1$
(3-23) $FN_2\lambda_2$
(3-24) $FN_3\lambda_3$
3-6 $-\Sigma FN_i\lambda_i$
MODULE 3
(6-3) $-(FL_1+FL_2)$
(7-25) $-FL_3$
3 ΣFL_i
(6-4) $-(FL_1b_1+FL_2b_2)$
(7-24) $-FL_3b_3$
4 ΣFL_i

1	3	3	15	A	1	15	B							
1	7	1	3	B	1	23	C							
1	16	1	7	B	4	11	N							
1	4	2	16	A	1	16	B							
1	8	1	4	B	1	22	C							
1	22	4	5											
1	24	1	22	S	1	7								
1	25	1	23	C	1	8								
1	15	1	8	B	4	11	N							
1	11	1	24	B										
1	10	1	25	B										
1	12	1	10	F										
1	13	1	10	F										
1	14	1	10	F										
1	19	1	13	B	4	22	N							
1	17	1	23	B	1	14	A							
1	18	1	22	B	1	14	A							
1	5	1	8	B	1	17	C							
1	6	1	18	B	1	7	C							
1	20	1	5	A										
1	21	1	6	B										
1	1	1	20	C	1	21	D							
4	22	1	1											
2	7	2	24	B										
2	9	2	7	B										
2	24	2	9	S	2	3								
2	5	2	9	B	4	18	A							
2	3	2	5	B	4	6	A							
2	10	2	25	B										
2	11	2	10	B										
2	25	2	11	C	2	4								
2	6	4	18	A	2	11	B							
2	4	4	6	A	2	6	B							
2	8	2	5	B	2	6	C							
2	15	2	8	A	3	3	D							
2	2	7	3	A	4	25	B	2	15	C				
2	16	2	2	A	3	10	D							
3	4	1	13	A										
3	3	3	4	B										
3	2	3	3	C	3	4	D							
3	10	3	2	B										
3	5	3	16	B	3	17	C	3	18	D				
3	1	3	2	C	4	19	B	3	5	D				
3	15	3	1	A	3	10	B							
4	12	4	9	F										
4	13	4	9	F										
4	14	4	9	F										
4	6	4	13	B	4	10	C	4	5	D				
4	5	4	14	B	4	10	C							
4	1	7	4	A	3	6	B	4	17	D	4	24	C	
4	20	4	1	A										
4	21	3	13	D										
4	11	4	20	B										
4	10	4	11	B										

Table 8-1. Connection Statements for Voyager Problem
(First of three sheets)

4	23	4	11	F										
4	17	2	15	N	3	12	D							
3	12	1	19											
3	13	1	19											
3	14	1	19											
2	12	1	19											
2	13	1	19											
2	14	1	19											
4	18	3	12	D	4	12	A							
2	19	3	14	N	4	11	D							
2	17	2	12	B	1	5	N							
2	1	2	19	C	2	17	D							
2	18	2	13	B	1	5	N							
4	24	2	1	X	4	16	S							
4	25	2	18	Y	4	16	C							
4	15	1	1	D	8	12	N							
4	19	2	14	N	4	15	B							
4	16	4	22	B	8	12	N							
8	12	1	10											
5	12	4	9											
5	15	4	20	N	5	12	B							
5	17	4	23	N	5	11	B							
5	5	1	4	A	5	15	B	5	17	C	5	19	D	
5	13	4	9											
5	19	5	13	A	5	11	B							
5	7	5	5	B	5	19	C							
5	11	5	7	B										
5	20	5	7	A										
5	21	4	11	D										
5	16	5	11	B	4	20	N							
5	18	5	12	B	4	23	N							
5	6	1	3	A	5	20	B	5	16	C	5	18	D	
5	14	4	9											
5	24	5	12	S	5	25								
5	25	5	5	C	5	14								
3	16	5	6	N	5	14	D							
3	19	3	16	B	5	11	N							
6	12	4	9											
6	15	4	20	N	6	12	B							
6	17	4	23	N	6	11	B							
6	5	1	4	A	6	15	B	6	17	C	6	19	D	
6	13	4	9											
6	19	6	13	A	6	11	B							
6	7	6	5	B	6	19	C							
6	11	6	7	B										
6	20	6	7	A										
6	21	4	11	D										
6	16	6	11	B	4	20	N							
6	18	6	12	B	4	23	N							
6	6	1	3	A	6	20	B	6	16	C	6	18	D	
6	14	4	9											
6	24	6	12	S	6	25								
6	25	6	5	C	6	14								
3	17	6	6	N	6	14	D							

Table 8-1 (Continued)

3	24	3	17	S	6	11	X					
6	3	5	25	A	6	25	B					
6	4	5	24	A	6	24	B					
7	12	4	9									
7	15	4	20	N	7	12	B					
7	17	4	23	N	7	11	B					
7	5	1	4	A	7	15	B	7	17	C	7	19 D
7	13	4	9									
7	19	7	13	A	7	11	B					
7	7	7	5	B	7	19	C					
7	11	7	7	B								
7	20	7	7	A								
7	21	4	11	D								
7	16	7	11	B	4	20	N					
7	18	7	12	B	4	23	N					
7	6	1	3	A	7	20	B	7	16	C	7	18 D
7	14	4	9									
7	24	7	12	S	7	25						
7	25	7	5	C	7	14						
3	18	7	6	N	7	14	D					
3	25	3	18	C	7	11	Y					
7	3	6	3	A	7	25	B					
7	4	6	4	A	7	24	B					
3	6	3	19	B	3	23	C	3	24	D		

Table 8-1 (Continued)

INTERCONNECTION MATRIX

I	J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	0	0	0	0	0	-6	-6	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	2	0	0	0	0	0	0	0	0	0	-10	0	0	0	0	0	0
6	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-2. External Matrix Assignments, Voyager Problem (first of eight sheets)

1	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-4	-4	
2	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
6	3	0	0	0	0	0	0	0	-8	0	0	0	0	0	0	0	0	
7	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
8	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
9	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
10	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
11	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
12	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
13	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
14	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
15	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
16	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
17	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
18	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
19	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
20	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-3	-3	
2	4	-7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	4	-9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
6	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
7	4	-10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
8	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
9	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
10	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
11	4	0	0	0	0	0	-4	0	0	0	0	0	0	0	0	0	0	
12	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
13	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
14	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
15	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
16	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
17	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
18	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
19	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
20	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Table 8-2 (continued)

1	5	0	0	0	0	0	0	0	0	-4	0	0	0	0	0	0	0	0
2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	5	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0
6	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	5	0	0	0	0	0	0	0	0	0	0	0	0	-2	0	0	0	0
8	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	5	-5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	6	0	0	0	0	0	0	0	0	0	0	0	0	-6	0	0	0	0
13	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-2 (continued)

1	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-9	0
7	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	7	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0
11	7	0	0	0	0	0	0	0	0	0	-7	0	0	0	0	0	0
12	7	0	0	0	0	0	0	0	0	0	0	0	0	-4	0	0	0
13	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	8	0	0	0	0	0	-8	0	0	0	0	0	0	0	0	0	0
5	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-6	0
7	8	0	-5	-5	0	0	0	0	0	0	0	0	0	0	0	0	0
8	8	0	0	0	0	0	0	0	0	0	0	0	0	-3	0	0	0
9	8	0	0	0	0	0	0	0	0	0	-9	0	0	0	0	0	0
10	8	0	0	0	0	0	0	0	0	0	0	0	0	0	-10	0	0
11	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-2 (continued)

1	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	9	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	9	0	0	0	0	-6	0	-2	0	0	0	0	0	0	0	0	0
7	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	9	0	0	0	0	0	0	0	0	0	0	0	0	-7	0	0	0
9	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	9	0	0	0	0	0	0	0	0	-9	0	0	0	0	0	0	0
13	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	10	0	0	-5	0	0	0	0	0	0	0	0	0	0	0	0	0
2	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	10	0	0	0	0	0	0	0	0	-7	0	0	0	0	0	0	0
4	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	10	0	-1	0	0	-1	0	0	0	0	-1	0	-1	0	0	0	0
8	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	10	0	0	0	0	0	0	0	0	-6	0	0	0	0	0	0	0
11	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-2 (continued)

1	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	11	0	0	0	0	0	0	-10	0	0	0	0	0	0	0	0
3	11	0	0	0	0	0	0	0	0	0	0	0	0	-5	0	0
4	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	11	0	0	0	0	0	0	0	0	0	0	0	0	0	-8	0
6	11	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	12	0	0	0	0	0	0	0	0	-1	0	-1	0	0	0	0
7	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-2 (continued)

1	13	0	0	0	0	0	0	0	0	0	0	0	0	-8	0	0	0
2	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	13	0	0	0	0	0	0	0	-6	0	0	0	0	0	0	0	0
4	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	13	0	0	0	0	0	-5	0	0	0	0	0	0	0	0	0	0
8	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0
7	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-2 (continued)

1	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	15	0	0	0	0	0	-10	0	0	0	0	0	0	0	0	0	0
5	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	15	0	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	0
9	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	15	0	0	0	0	-3	0	0	0	0	0	0	0	0	0	0	0
11	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	15	0	0	0	0	-4	0	0	0	0	0	0	0	0	0	0	0
13	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

IS	JS	KS	LS	MSMAX	ALG	MS
20	15	16	0	20	3	10

MIDDLE BLOCKS ASSIGNED

J=	1	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J=	2	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J=	3	8	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J=	4	7	9	10	4	3	0	0	0	0	0	0	0	0	0	0	0
J=	5	5	4	1	2	0	0	0	0	0	0	0	0	0	0	0	0
J=	6	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J=	7	7	4	1	9	0	0	0	0	0	0	0	0	0	0	0	0
J=	8	5	8	9	3	10	6	0	0	0	0	0	0	0	0	0	0
J=	9	1	6	2	9	7	0	0	0	0	0	0	0	0	0	0	0
J=	10	1	3	7	6	0	0	0	0	0	0	0	0	0	0	0	0
J=	11	2	10	5	8	0	0	0	0	0	0	0	0	0	0	0	0
J=	12	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J=	13	5	6	8	0	0	0	0	0	0	0	0	0	0	0	0	0
J=	14	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J=	15	2	3	4	10	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-2 (continued)

INTERCONNECTION MATRIX

I	J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	1	0	0	0	-4	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	2	-3	-1	0	-3	0	0	0	0	0	0	0	0	0	0	0	0	0
3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	2	0	0	0	0	0	0	-4	0	0	0	0	0	0	0	0	0	0
5	2	0	-2	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0
1	3	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0
2	3	0	-3	-3	0	0	-3	0	0	0	0	0	0	0	0	0	0	0
3	3	0	-4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	3	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0
5	3	0	-5	0	0	-5	0	-5	0	0	0	0	0	0	0	0	0	0
1	4	0	0	0	-5	0	0	0	0	0	0	0	0	0	0	0	0	0
2	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	4	-4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	4	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	4	0	0	0	0	0	0	-2	0	0	0	0	0	0	0	0	0	0
1	5	0	0	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	0
2	5	0	0	0	0	0	-4	0	0	0	0	0	0	0	0	0	0	0
3	5	0	0	-5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	5	0	0	0	0	-3	0	0	0	0	0	0	0	0	0	0	0	0

IS	JS	KS	LS	MSMAX	ALG	MS
5	5	7	1	15	3	5

MIDDLE BLOCKS ASSIGNED

J= 1	4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 2	3	1	2	4	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 3	3	4	5	2	1	0	0	0	0	0	0	0	0	0	0	0	0
J= 4	4	1	5	2	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 5	5	3	2	4	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-3. Internal Matrices for Voyager Problem
(first of seven sheets)

INTERCONNECTION MATRIX

I	J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	-3	0	-3	0	0	0	0	0	0	0	0	0	0
3	1	0	0	-1	0	0	0	-2	0	0	0	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	2	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0
3	2	0	-4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	3	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	3	0	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	3	0	0	0	-4	0	0	0	0	0	0	0	0	0	0	0	0	0
5	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	4	0	0	0	0	0	-4	0	0	0	0	0	0	0	0	0	0	0
2	4	0	-2	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	4	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	5	0	0	-3	-3	0	0	0	0	0	0	0	0	0	0	0	0	0
2	5	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0
3	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	5	0	0	-4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	5	0	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0

IS	JS	KS	LS	MSMAX	ALG	MS
5	5	7	2	15	3	4

MIDDLE BLOCKS ASSIGNED

J= 1	1	3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 2	3	4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 3	1	2	3	4	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 4	2	1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 5	3	4	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-3 (continued)

INTERCONNECTION MATRIX

I	J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	0	0	-2	0	-2	0	0	0	0	0	0	0	0	0	0	0	0
2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	2	0	-3	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	3	0	-2	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0
5	3	0	0	0	-3	0	0	0	0	0	0	0	0	0	0	0	0	0
1	4	0	0	-4	-4	0	0	0	0	0	0	0	0	0	0	0	0	0
2	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	4	0	0	0	0	-5	0	-5	0	0	0	0	0	0	0	0	0	0
5	4	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0
1	5	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	5	0	-4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	5	0	0	-5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

IS	JS	KS	LS	MSMAX	ALG	MS
5	5	7	3	15	3	5

MIDDLE BLOCKS ASSIGNED

J= 1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 2	3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 3	2	1	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 4	4	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 5	4	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-3 (continued)

INTERCONNECTION MATRIX

I	J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	0	0	0	0	0	-3	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	0	0	0	-2	0	-1	0	0	0	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	2	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	2	0	-2	0	0	0	0	-2	0	0	0	0	0	0	0	0	0	0
5	2	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	3	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	3	0	0	0	-3	0	0	0	0	0	0	0	0	0	0	0	0	0
5	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	4	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0
3	4	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	4	0	0	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	5	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0
2	5	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0
3	5	0	0	0	0	0	0	-3	0	0	0	0	0	0	0	0	0	0
4	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

IS	JS	KS	LS	MSMAX	ALG	MS
5	5	7	4	15	3	3

MIDDLE BLOCKS ASSIGNED

J= 1	2	3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 2	3	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 3	1	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 4	2	3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 5	2	1	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-3 (continued)

INTERCONNECTION MATRIX

I	J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	2	0	0	0	0	0	-4	-4	0	0	0	0	0	0	0	0	0	
3	2	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	2	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	
5	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	3	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	3	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	
5	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	4	0	0	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	
4	4	0	0	0	0	-4	0	0	0	0	0	0	0	0	0	0	0	
5	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	5	0	0	0	-3	0	0	-3	0	0	0	0	0	0	0	0	0	
2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	5	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	5	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	
5	5	-4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

IS	JS	KS	LS	MSMAX	ALG	MS
5	5	7	5	15	3	4

MIDDLE BLOCKS ASSIGNED

J= 1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
J= 2	2	1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
J= 3	3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
J= 4	4	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
J= 5	4	2	3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	

Table 8-3 (continued)

INTERCONNECTION MATRIX

I	J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	1	0	-4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	2	0	0	0	0	0	-4	-4	0	0	0	0	0	0	0	0	0	
3	2	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	2	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	
5	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	3	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	3	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	
5	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	4	0	0	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	
4	4	0	0	0	0	-3	0	0	0	0	0	0	0	0	0	0	0	
5	4	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	
1	5	0	0	0	-3	0	0	-3	0	0	0	0	0	0	0	0	0	
2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	5	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	5	0	0	0	0	-4	0	0	0	0	0	0	0	0	0	0	0	
5	5	-2	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	

IS	JS	KS	LS	MSMAX	ALG	MS
5	5	7	6	15	3	4

MIDDLE BLOCKS ASSIGNED

J= 1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 2	2	1	4	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 3	3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 4	3	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 5	2	1	3	4	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-3 (continued)

INTERCONNECTION MATRIX

I	J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	1	0	-4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	2	0	0	0	0	0	-4	-4	0	0	0	0	0	0	0	0	0	
3	2	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	2	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	
5	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	3	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	3	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	
5	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	4	0	0	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	
4	4	0	0	0	0	-3	0	0	0	0	0	0	0	0	0	0	0	
5	4	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	
1	5	0	0	0	-3	0	0	-3	0	0	0	0	0	0	0	0	0	
2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	5	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	5	0	0	0	0	-4	0	0	0	0	0	0	0	0	0	0	0	
5	5	-2	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	

IS	JS	KS	LS	MSMAX	ALG	MS
5	5	7	7	15	3	4

MIDDLE BLOCKS ASSIGNED

J= 1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 2	2	1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 3	3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 4	3	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 5	2	1	3	4	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-3 (continued)

This program, as originally furnished by NASA (Reference 4) used a large number of non-standard components, such as a feedback limiter built with a battery and a a number of resistors, diodes, and capacitors in conjunction with operational amplifiers for generating pulses of certain specified duration, delaying these pulses, etc. For implementing on a system such as the proposed one, these must be replaced by standard analog and logic circuits.

The "battery limiter" was used in place of the conventional feedback limiter because of its flat limiting characteristic. Examination of the description of this limiter indicates that its limit is not as flat as that of a modern feedback limiter, which contains active circuitry. Hence the regular feedback limiters may be used. The pulse and timing circuits were replaced by gates, flip/flops and monostable timers. For reasons given in Chapter 7, the logic connections are not included in the list of connection statements.

Since the problem requires six analog modules, it could be run on the single-console system except for the need for six resolvers. The problem was programmed on the two-console system using the six odd-numbered modules (the resolver modules) and the results are included in the tabulation given in Section 8.3.

The problem was also programmed on the single-console system, using variable DFG's for the extra sine/cosine functions. This gives a very dense one-console problem; that is, most of the equipment in all six modules is used. Only the single-console program is presented in full in this report.

The program used essentially the same equations as the original program provided by NASA, but the equations involving axis rotation were written in a different form. The difference is in the grouping of the terms. The original equations were programmed on servo multipliers and resolvers, which places strong emphasis on arranging the equations in terms of products with common factors. When the problem is programmed with quarter-square multipliers, the emphasis is on reducing the total number of multiplications, since two products with a common factor use as much equipment as two products with completely different factors.

In re-grouping the equations, I noticed that terms such as $\sin\theta_z \cos X_z - \cos\theta_z \sin X_z$ were needed. This is equivalent to $\sin(\theta_z - X_z)$, and if it is generated by subtracting X_z from θ_z, several multipliers are saved. I deliberately did not do this in the program given in this chapter, since I wanted to use all six modules as fully as possible in order to test the system capability.

To aid the programmer in following the diagram, the re-grouped rotational equations are given below:

$$\ddot{\phi}_{ygp}/G_g = \sin\left[2(\theta_z - X_z)\cos\theta_x\right]\cos\theta_y$$

$$-\ddot{\phi}_{gqy}/G_g = \sin\left[2(\theta_z - X_z)\cos\theta_x\right]\sin\theta_y$$

$$\dot{\phi}_{ap}/A_a = \left[\sin(\theta_z - X_z)\cos\theta_x\right]\cos\theta_y + \left[\cos(\theta_z - X_z)\sin\theta_x\right]\sin\theta_y + \left[\sin(\theta_z - X_z)\sin\theta_x\right]\cos\theta_y$$

$$\dot{\phi}_{ay}/A_a = \left[\sin(\theta_z - X_z)\cos\theta_x\right](-\sin\theta_y) + \left[\cos(\theta_z - X_z)\sin\theta_x\right]\cos\theta_y + \left[\sin(\theta_z - X_z)\sin\theta_x\right]\sin\theta_y$$

$$\psi_y = \left[\sin(X_z - \theta_z)\cos\theta_x\right]\sin X_y - \left[\sin\theta_x\right]\cos X_y$$

$$-\psi_p = \left[\sin(X_z - \theta_z)\cos\theta_x\right]\cos X_y + \left[\sin\theta_x\right]\sin X_y$$

$$\psi_R = Q\cos X_y - \left[\cos\theta_x\cos\theta_y\right]\sin X_y$$

where $Q = \left[\sin(\theta_z + X_z)\sin\theta_x\right]\cos\theta_y + \cos(\theta_z - X_z)\sin\theta_y$

Figure 8-2 gives the circuit diagram, and Tables 8-4 through 8-6 give the connection statements and the printout of the programming array.

The external matrix program required all 8 middle blocks, according to the algorithm. This was not due to the heavy matrix traffic, but due to a deficiency in the algorithm. With a little re-assignment, I was able to eliminate two middle blocks. Hence this program, like all other single-console programs studied in this project, requires only six middle blocks (one of the others took only five; see Section 8.3).

I think I know how to modify the algorithm to improve its performance on problems like this one, but this would require additional programming effort and re-compiling. The important thing is to demonstrate the feasibility of the switching network, and the programming array in Figure 8-5 does this.

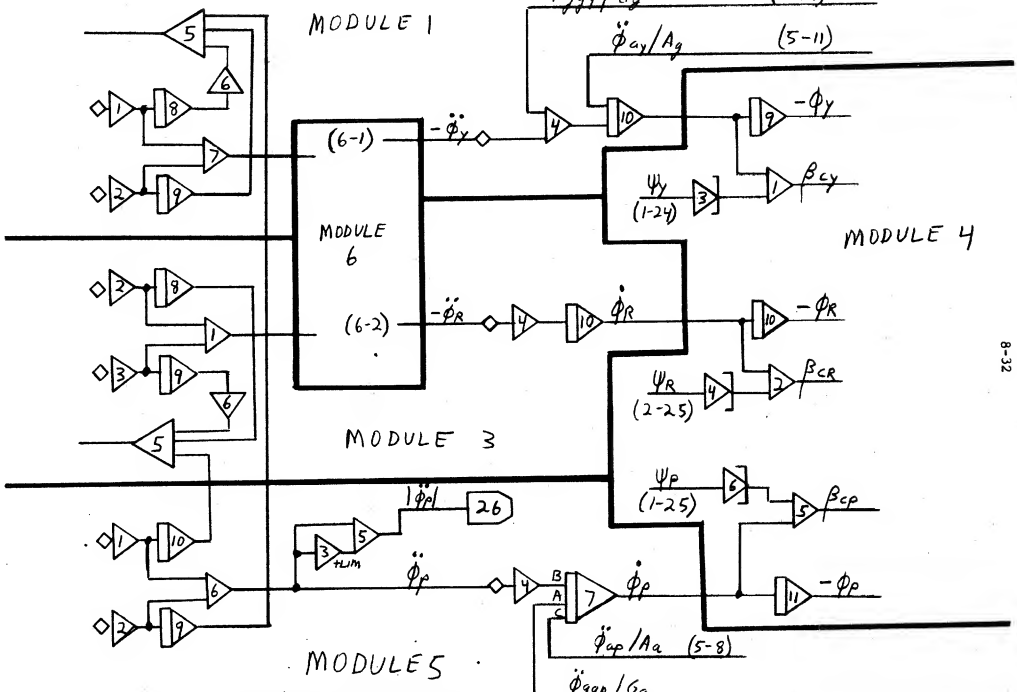


Figure 8.2 Circuit Diagram for S-IV, Phase 3 (First of four sheets)

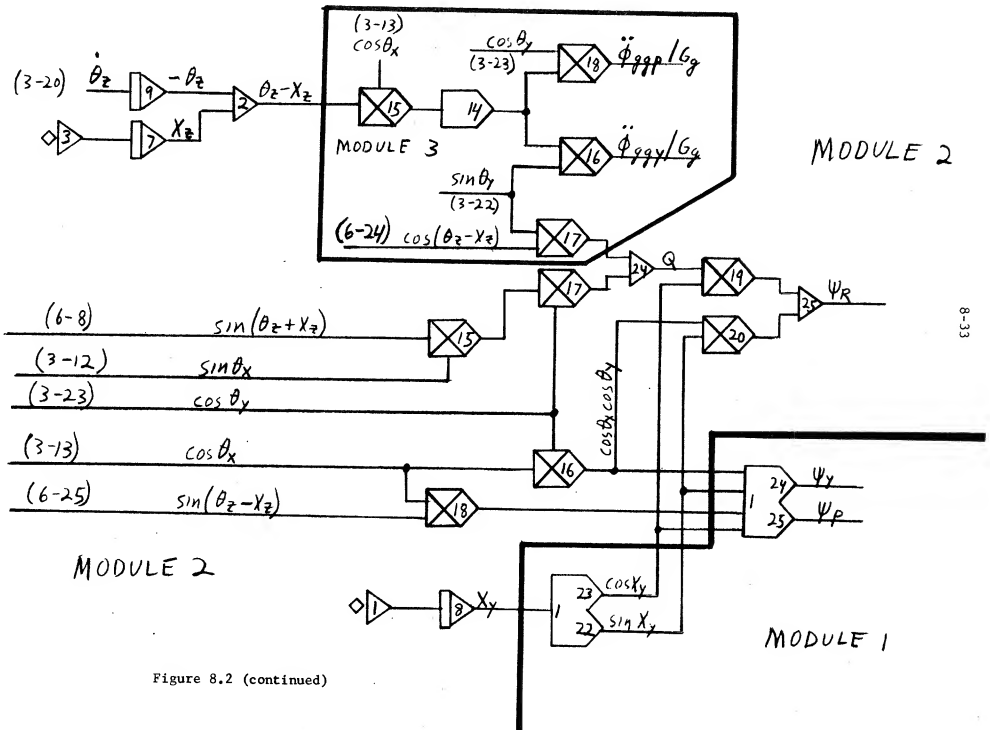
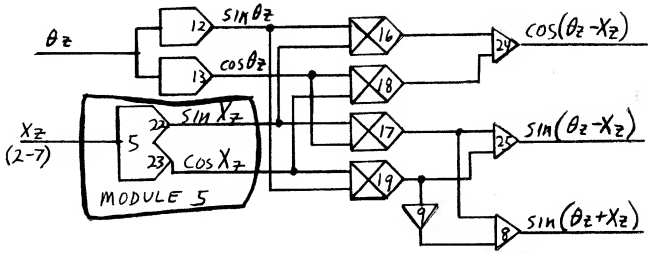
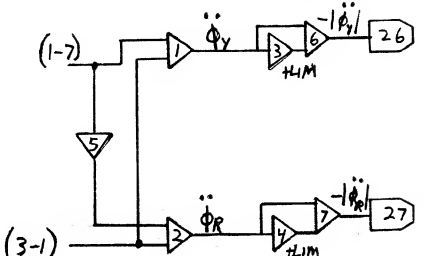


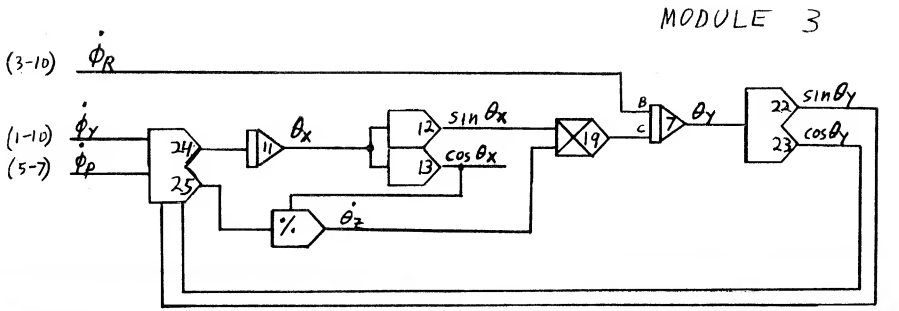
Figure 8.2 (continued)





MODULE 6

Figure 8.2 (continued)



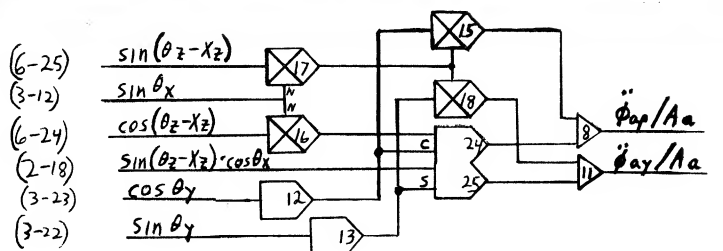


Figure 8.2 (continued) DFG's 12 and 13 are needed only because "S" and "C" inputs must be internal

MODULE 5

1	8	1	1	B								
1	9	1	2	B								
1	7	1	1	B	1	2	C					
1	6	1	8	B								
1	5	1	6	B	1	9	C	5	9	A		
1	4	1	3	B	6	1	A					
1	10	1	4	B	5	11	A					
4	9	1	10	A								
4	3	1	24	A								
4	1	4	3	C	1	10	A					
3	8	3	2	B								
3	9	3	3	B								
3	1	3	2	C	3	3	D					
3	4	6	2	A								
3	10	3	4	B								
4	10	3	10	A								
4	2	3	10	A	4	4	C					
4	4	2	25	A								
3	6	3	9	B								
3	5	3	6	B	3	8	C	5	10	A		
5	10	5	1	B								
5	9	5	2	B								
5	6	5	1	B	5	2	C					
5	4	5	6	B								
5	7	5	4	B	3	18	A	5	8	C		
4	11	5	7	A								
4	6	1	25	A								
4	5	5	7	A	4	6	B					
2	9	3	20	A								
2	2	2	9	C	2	7	D					
2	7	2	3	B								
3	15	2	2	N	3	13	B					
3	14	3	15	F								
3	18	3	23	A	3	14	B					
3	16	3	14	A	3	22	B					
5	22	2	7									
3	17	3	22	A	6	24	D					
2	15	6	8	N	3	12	D					
2	17	2	15	B	3	23	N					
2	24	3	17	X	2	17	S					
2	19	2	24	A	1	23	D					
2	16	3	13	D	3	23	N					
5	3	5	6	B								
5	5	5	3	B	5	6	C					
5	26	5	5									
2	20	2	16	A								
2	21	1	22	D								
2	25	2	19		2	20	C					
2	18	6	25	N	3	13	D					
1	24	2	16	X	1	22	S					
1	25	2	18	Y	1	23	C					
2	8	2	1	B								
1	22	2	8									
3	24	1	10	X	3	22	S					

Table 8-4. Connection Statements for S-IV Problem, Phase 3. (first of two sheets)

3	25		5	7	Y		3	23	C
3	11		3	24	B				
3	12		3	11	F				
3	13		3	11	F				
3	19		3	12	A		3	20	B
3	20		3	25	A				
3	21		3	13	B				
3	19		3	12	A		3	20	B
3	7		3	19	C		3	10	B
3	22		3	7	F				
5	17		5	22	B		3	12	N
5	16		5	23	B		3	12	N
5	12		3	23					
5	13		3	22					
5	15		5	17	A		5	12	B
5	18		5	17	A		5	13	B
5	24		5	16			5	12	S
5	25		3	18	Y		5	13	C
5	8		5	15	B		5	24	C
5	11		5	18	B		5	25	A
6	12		2	7					
6	13		2	7					
6	16		6	12	B		5	22	N
6	18		6	13	B		5	23	N
6	17		6	13	B		5	22	N
6	19		6	12	B		5	23	N
6	24		6	16			6	18	S
6	25		6	17			6	19	C
6	9		6	19	B				
6	8		6	17	B		6	9	C
6	5		1	7	A				
6	1		1	7	A		3	1	B
6	2		6	5	C		3	1	B
6	3		6	1	B				
6	6		6	1	B		6	3	C
6	26		6	6					
6	4		6	2	B				
6	7		6	2	B		6	4	C
6	27		6	7					

Table 8-4 (continued)

INTERCONNECTION MATRIX

I	J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	1	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	1	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0
8	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	2	0	-X4	-X4	0	0	0	0	0	0	0	0	0	0	0	0	0
3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	2	0	0	0	0	0	0	0	-X3	0	0	0	0	0	0	0	0
5	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	0	-X6	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	3	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0
5	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	3	0	0	0	0	0	0	-3	0	0	0	0	0	0	0	0	0

Table 8-5 External Matrix Connections for S-IV, Phase 3
(first of four sheets)

1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	4	0	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	0
3	4	0	0	0	-4	0	0	0	0	0	0	0	0	0	0	0	0
4	4	0	0	0	0	0	0	0	0	0	0	0	-X3	0	0	0	0
5	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	4	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0
11	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	4	0	0	0	-5	0	0	0	0	0	0	0	0	0	0	0	0
13	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	5	0	0	0	0	-6	0	0	0	0	0	0	0	0	0	0	0
2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	5	0	0	0	0	0	0	0	0	0	0	-X4	0	0	0	0	0
4	5	0	0	0	0	0	0	0	0	0	0	-3	0	0	0	0	0
5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	5	0	-X1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	6	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0
3	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	6	0	0	0	0	0	0	-X6	0	0	-X6	0	0	0	0	0	0
9	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	6	0	0	0	-3	-3	0	0	0	0	0	0	0	0	0	0	0
14	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-5 (continued)

1	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	7	0	0	0	0	-5	0	0	0	0	0	0	0	0	0	0	0
3	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	7	0	0	0	0	0	0	0	0	0	0	-X2	0	0	0	0	0
9	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	7	0	0	0	0	0	0	0	0	-X1	0	0	-X1	0	0	0	0
11	7	-X6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	7	0	0	0	0	0	-X4	0	-X4	0	0	0	-X4	0	0	0	0
15	7	0	0	-X3	0	0	0	0	0	0	0	0	0	0	0	0	0

1	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	8	0	0	0	0	0	0	0	-6	-X6	0	0	0	0	0	0	0
13	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1	9	0	0	0	0	0	0	0	0	0	-X5	-X5	0	0	0	0	0
2	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	9	0	0	0	0	0	0	0	0	0	0	-X6	0	0	0	0	0
8	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	9	0	0	0	0	0	0	0	0	0	0	0	-X2	0	0	0	0
11	9	0	0	0	0	-X1	0	0	-X1	-X4	0	0	0	0	0	0	0
12	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-5 (continued)

1	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	10	0	0	0	0	0	0	0	0	0	0	0	-X6	0	0	0
11	10	0	-X3	0	0	0	0	0	0	-X3	0	0	0	0	0	0
12	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

IS	JS	KS	IS	MSMAA	ALG	MS
15	10	12	0	15	3	8

MIDDLE BLOCKS ASSIGNED

J=	1	3	2	1	0	0	0	0	0	0	0	0	0	0	0	0
J=	2	8	4	0	0	0	0	0	0	0	0	0	0	0	0	0
J=	3	7	1	3	0	0	0	0	0	0	0	0	0	0	0	0
J=	4	4	5	2	1	8	0	0	0	0	0	0	0	0	0	0
J=	5	4	6	7	3	0	0	0	0	0	0	0	0	0	0	0
J=	6	2	3	8	0	0	0	0	0	0	0	0	0	0	0	0
J=	7	2	7	5	1	6	8	0	0	0	0	0	0	0	0	0
J=	8	6	5	0	0	0	0	0	0	0	0	0	0	0	0	0
J=	9	7	1	4	3	0	0	0	0	0	0	0	0	0	0	0
J=	10	6	2	5	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-5 (continued)

INTERCONNECTION MATRIX

I	J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	
2	1	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	2	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	
2	2	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	2	0	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	3	0	0	0	0	0	0	-2	0	0	0	0	0	0	0	0	0	
2	3	0	0	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	3	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	
1	4	0	0	0	0	-3	0	0	0	0	0	0	0	0	0	0	0	
2	4	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	
3	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

IS	JS	KS	LS	MSMAX	ALG	MS
5	5	7	1	15	3	3

MIDDLE BLOCKS ASSIGNED

J=	1	1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
J=	2	3	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	
J=	3	3	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	
J=	4	1	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
J=	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Table 8-6. Internal Matrices for S-IV, Phase 3. (first of six sheets)

INTERCONNECTION MATRIX

I	J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	2	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	
3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	2	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	
5	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	3	0	0	0	-3	0	0	0	0	0	0	0	0	0	0	0	0	
2	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	3	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	4	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	4	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	5	0	0	0	0	0	0	-3	0	0	0	0	0	0	0	0	0	
4	5	0	0	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	
5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

IS	JS	KS	LS	MSMAX	ALG	MS
5	5	7	2	15	3	3

MIDDLE BLOCKS ASSIGNED

J= 1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
J= 2	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
J= 3	1	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
J= 4	3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
J= 5	2	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Table 8-6 (continued)

INTERCONNECTION MATRIX

I	J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	1	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	1	0	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	2	0	0	-3	-3	0	0	0	0	0	0	0	0	0	0	0	0	
2	2	0	0	0	0	0	-4	0	0	0	0	0	0	0	0	0	0	
3	2	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	2	-2	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	3	0	0	0	-2	-2	0	0	0	0	0	0	0	0	0	0	0	
2	3	0	0	0	-4	0	0	0	0	0	0	0	0	0	0	0	0	
3	3	0	-5	0	0	0	0	-5	0	0	0	0	0	0	0	0	0	
4	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	3	-3	0	0	0	0	0	-3	0	0	0	0	0	0	0	0	0	
1	4	0	0	0	0	-5	0	0	0	0	0	0	0	0	0	0	0	
2	4	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	4	-4	-4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	4	0	0	0	0	0	0	-2	0	0	0	0	0	0	0	0	0	
1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	5	0	0	0	-5	0	0	0	0	0	0	0	0	0	0	0	0	
3	5	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	5	0	0	0	0	-3	0	0	0	0	0	0	0	0	0	0	0	
5	5	0	0	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	

IS	JS	KS	LS	MSMAX	ALG	MS
5	5	7	3	15	3	5

MIDDLE BLOCKS ASSIGNED

J= 1	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	
J= 2	1	2	3	4	0	0	0	0	0	0	0	0	0	0	0	
J= 3	3	5	2	4	0	0	0	0	0	0	0	0	0	0	0	
J= 4	4	1	5	2	0	0	0	0	0	0	0	0	0	0	0	
J= 5	1	5	3	2	0	0	0	0	0	0	0	0	0	0	0	

Table 8-6 (continued)

INTERCONNECTION MATRIX

I	J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	1	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	3	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	4	0	0	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

IS	JS	KS	LS	MSMAX	ALG	MS
5	5	7	4	15	3	3

MIDDLE BLOCKS ASSIGNED

J= 1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 4	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
J= 5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8-6 (continued)

INTERCONNECTION MATRIX

I	J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	0	0	-1	0	-1	0	0	0	0	0	0	0	0	0	0	0	
2	1	0	0	-2	-2	0	0	-2	0	0	0	0	0	0	0	0	0	
3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	2	0	0	0	-4	-4	0	0	0	0	0	0	0	0	0	0	0	
2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	2	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	2	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	2	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	3	0	0	-5	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	3	0	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	
3	3	0	-3	0	0	0	0	-3	0	0	0	0	0	0	0	0	0	
4	3	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	3	0	-4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	4	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	
2	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	4	0	0	0	0	0	0	-4	0	0	0	0	0	0	0	0	0	
5	4	0	0	0	0	-5	0	0	0	0	0	0	0	0	0	0	0	
1	5	0	0	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

IS	JS	KS	LS	MSMAX	ALG	MS
5	5	7	5	15	3	5

MIDDLE BLOCKS ASSIGNED

J= 1	1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
J= 2	3	1	2	4	0	0	0	0	0	0	0	0	0	0	0	0	
J= 3	1	3	4	5	2	0	0	0	0	0	0	0	0	0	0	0	
J= 4	1	5	4	0	0	0	0	0	0	0	0	0	0	0	0	0	
J= 5	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Table 8-6 (continued)

INTERCONNECTION MATRIX

I	J	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	0	0	-4	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	1	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	2	0	0	0	-3	0	0	-3	0	0	0	0	0	0	0	0	0	
2	2	0	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	
3	2	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	2	-4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	3	0	0	0	-4	0	0	0	0	0	0	0	0	0	0	0	0	
2	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	3	0	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	3	0	0	0	-1	0	-1	0	0	0	0	0	0	0	0	0	0	
5	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	4	0	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	
2	4	0	0	0	0	-3	0	0	0	0	0	0	0	0	0	0	0	
3	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	4	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	5	0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	5	0	0	0	0	-4	0	-4	0	0	0	0	0	0	0	0	0	
5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

IS	JS	KS	LS	MSMAX	ALG	MS
5	5	7	6	15	3	4

MIDDLE BLOCKS ASSIGNED

J= 1	4	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
J= 2	4	2	3	1	0	0	0	0	0	0	0	0	0	0	0	0	
J= 3	3	4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	
J= 4	1	2	3	0	0	0	0	0	0	0	0	0	0	0	0	0	
J= 5	1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Table 8-6 (continued)

8.3 Summary of Problems Programmed

In all, five problems furnished by NASA were programmed on the proposed systems. They ranged from fairly small and almost linear problems (the heat-flux partial differential equation) to moderately large and highly nonlinear ones, such as the two presented in this chapter. One of the problems was programmed in two different ways (as a two-console problem with six resolvers, and as a one-console problem with variable DFG's for some of the sine-cosine generation). This gave a total of six programs: two on the large system (two-consoles) and four on the small system (one console).

Table 8-7 lists the results. This table gives, for each program, the number of modules required, the number of consoles, the number of middle blocks on the external matrix, and the maximum number of middle blocks on the internal matrices.

The following points should be noted:

(a) Both two console problems required only 10 (rather than 12) middle blocks on the external matrix.

(b) All one-console problems took 6 (rather than 8) middle blocks on the external matrix, except for the mass cable problem, which took only five.

(c) No problem required more than 5 middle blocks on the internal matrix, and one (the heat-flux problem) took only four.

Note that these results were achieved without any component interchanging to equalize the traffic on the input and output blocks. Perhaps we could do even better if an interchanging algorithm were included. Thus we have fairly strong evidence that we could probably reduce the number of middle blocks in each matrix by 1 or 2, thus reducing the total switch count by 10% to 20%. Exactly how much we could reduce the matrices is hard to determine. A large number of analog problems, rather than a small sample, would have to be examined. A proper optimization of the system would include actually writing the software for sectioning the problems and assigning components, and programming dozens of different problems, but the problems studied so for indicate that the proposed system is adequate, and probably can be reduced further.

Additional evidence for this contention may be seen in the study of twenty randomly-generated program with typical fanout distributions given in Chapter 9.

Problem Name	Modules	Consoles	Middle Blocks in External Matrix	Max. Middle Blocks in Internal Matrices
Phase 1, S-IV Stage Control	5	1	6	5
Phase 3, S-IV Stage Control	6	2	10	5
Phase 3, S-IV Stage Control	6	1	6	5
Mass-cable System	5	1	5	5
Heat Flux (P.D.E.)	4	1	6	4
Voyager-First Stage Ascent	8	2	10	5

Table 8-7

Summary of NASA Sample Problems

9. COMPUTER RESULTS

In addition to programming sample problems on the proposed systems, a fairly extensive statistical evaluation was performed of the three-stage matrices. This chapter describes the algorithm used to program the matrices, the types of experimental cases run, and the results.

9.1 The Switch Assignment Algorithm

Given a program (defined by its programming array, as in Chapter 5), how should middle blocks be assigned to implement this program, and what is the minimum number of middle blocks required? Quite a bit of effort was expended developing an assignment algorithm. The current version of the algorithm has been tried on hundreds of programs and has been found fairly efficient. Ideally, an algorithm should implement every program with the minimum number of middle blocks, but it is not always easy to decide what the minimum number is. Suppose the algorithm takes eight middle blocks to implement a given program. How can we be sure that it can't be done with seven? Theoretically the problem is trivial; all we have to do is try all possible combinations. A fairly dense program on the external matrix of the single-console system may contain as many as 96 entries in its programming array (eight entries per column times twelve columns). If we want to see whether such a program can be implemented with seven middle blocks, we may try all possible middle block assignments. Each entry may be replaced by an integer from one to seven; hence there are 7^{96} ways of making these assignments. We can, in principle, examine each of these assignments to see if it satisfies the necessary conditions (each number is used at most once in a column, and each number is used in at most one row in any input block). If one of these assignments satisfies both conditions, we have found a way to solve the problem with seven middle blocks; if none of them does, we have proved that the problem cannot be solved with seven middle blocks. However, no sane person would attempt to examine all 7^{96} possibilities, even with a computer.

There is one obvious lower bound that is easy to calculate. By counting the entries in each column, we can determine the maximum "column length." The number of middle blocks must be at least this great. If, in fact, the algorithm does solve the problem with that many middle blocks, then we know that the solution is optimal (for that particular program). However, we have seen in Chapter 5 that sometimes it is necessary to use more middle blocks; the maximum column length is not a sufficient condition.

The current version of the algorithm has been tested on hundreds of "random output-full" programs (as defined in Section 9.2) and the following conclusions may be drawn:

a. In about half the cases, the number of middle blocks used was equal to the maximum column length. In these cases, the algorithm obviously produced optimum results.

b. In the rest of the cases, almost all of them used just one more middle block than the maximum column length. In these cases, the result may or may not have been optimal, but in any case, it wasn't far from it. I suspect that the "extra middle block was actually necessary in most of these cases, but this is fairly hard to prove (the methods of Chapter 5 are good for constructing arrays with maximal blocking, but very poor for analyzing any given array).

In any case, it is fairly clear that the algorithm is close enough to optimal for practical purposes.

This section describes the general idea of the algorithm without presenting the detailed flow chart, which is quite complicated. The algorithm proceeds column by column; that is, it makes all connections within a column before proceding to the next column. The first thing it does is to count the entries in each column and re-arrange them in order of decreasing column length, so that the worst columns are tackled first. (The computer printout, however, prints the columns in their original order.)

The first column can be assigned arbitrarily, since this can be considered as an "initial labeling" of the middle blocks.

Once one or more columns are assigned, certain possibilities are ruled out for making subsequent connections. In starting out with a new column, the first thing the algorithm does is to tabulate, for each connection in that column, the middle blocks that can't be used for that particular connection, because they have already been used for some other input in the same input block. The connections are then examined to see which one has the fewest remaining possible middle blocks. This is the most "urgent" connection, and the algorithm decides that this is the connection to be made first. The next step is to determine which middle block to use (assuming more than one is available). The one chosen is the one that will block the fewest future connections. For example, suppose a given connection has only two

possibilities left, namely middle block number 3 and middle block number 8. Suppose, on examining the tabulation of impossible connections, we discover that middle block number 3 is a possible candidate for three of the connections that we have yet to make in this column, while middle block number 8 is a possible candidate for five of these connections. We would make the connection with middle block number 3 on the grounds that it is the least useful for other purposes, and hence by using it, we are foreclosing the fewest future possibilities.

In short, the algorithm makes the most urgent connection first, and makes it with the least flexible middle block. After each connection is made, the number of possibilities for other connections is reduced, so that the table of impossible connections must be updated after each assignment.

There will be many "ties" in making such comparisons; that is, several connections may be equally urgent or several middle blocks equally flexible. In such cases, additional tests are made, based on such additional criteria as the fanout for the various inputs, the number of inputs in use within a given input block, and so on. These tests are fairly complicated and will not be covered here.

If a connection has no possibilities left, then the algorithm adds one more middle block and starts over again, reassigning the entire matrix from the beginning. The strategy of adding one more middle block and continuing without re-starting was also tried, but it turned out to give poorer results (more middle blocks required) than the "go back and start over" strategy. Even with restarting, the entire algorithm takes less than ten seconds on a large matrix (the external matrix for the two-console system).

The algorithm has been coded in FORTRAN IV and run on the 8400. It uses a lot of memory, chiefly because of the need to store the programming array. The large system, for example, has 300 matrix inputs and 16 matrix output blocks. Hence the programming array alone requires 4800 words of storage. This could be reduced considerably by "packing," since we don't need a full 32-bit word to store the middle-block address; but this packing is somewhat awkward in FORTRAN. For execution on a small computer, the algorithm may have to be coded in assembly language.

9.2 The Statistical Studies

In addition to programming actual problems furnished by NASA, the algorithm was used to implement randomly generated programs. Two

kinds of randomly generated programs were used: the random output-full programs and random fanout programs.

9.2.1 Random Output-Full Programs

A program will be called output-full if it uses every matrix output. Any legitimate program can be expanded to an output-full program merely by adding more connection statements. The expanded program is at least as hard to implement as the original, and probably harder. Hence a matrix capable of handling output-full programs is capable of handling all programs; put another way, the set of all output-full programs constitutes a "worst-case" set.

An N-by-M matrix has $(N+1)^M$ programs; of these, N^M are output-full. We cannot examine all of these for lack of time, but we can pick one at random simply by picking one of the N inputs at random to connect to output number 1, another to connect to output number 2, etc. Since M>N for any practical analog matrix (i.e. the matrix outputs outnumber the inputs), an output-full matrix will of necessity have quite a bit of fanout; in fact, if all matrix outputs are used, the average fanout is M/N; that is, it is equal to the expansion factor of the matrix, which is about two for most matrices.

If the algorithm is tried on a number of randomly-selected output-full programs, we should have a fairly good test of both the algorithm and the matrix. Accordingly, such tests were run on the following four matrices:

Matrix A: 50 inputs grouped in 10 blocks of 5; 96 outputs grouped in 12 blocks of 8. Expansion factor = 1.92. Eight middle blocks.

Matrix B: 105 inputs grouped in 15 blocks of 7; 192 outputs grouped in 16 blocks of 12. Expansion factor = 1.83. Twelve middle blocks.

Matrix C: 25 inputs grouped in 5 blocks of 5; 49 outputs grouped in 7 blocks of 7. Expansion factor = 1.96. Seven middle blocks.

Matrix D: 27 inputs grouped in 9 blocks of 3; 50 outputs grouped in 10 blocks of 5. Expansion factor = 1.85. Five middle blocks.

Matrices A and B are not really part of the proposed system. They are based on the maximum expected traffic on the single-console and double-console system respectively. As pointed out in Chapters 6 and 7, the recommended external matrix for the single-console system was

obtained by enlarging the input blocks of Matrix A to 15 inputs each and the output blocks of Matrix A to 16 outputs each. This enlarged matrix obviously requires at least 16 middle blocks to handle all possible programs; but remember that we are assuming two things: first, that the traffic on the external matrix is light; and second, that we can interchange components to distribute the traffic equally. We intend to use no more than eight outputs in any one output block and five inputs in any one input block. Any program that satisfies these restrictions is equivalent to a program on Matrix A. In fact, all we have to do is delete 10 unused inputs in each input block and renumber the remaining ones 1 through 5. This gives us a programming array on Matrix A. Hence the study of arbitrary programs on matrix A is equivalent to the study of programs on the actual recommended matrix which use at most five inputs per input block and eight outputs per output block.

Similar remarks hold for Matrix B which is the basis for the design of the two-console system.

Both Matrix A and Matrix B were designed by the formulas in Chapter 5. By this theory, eight middle blocks should be adequate for A and twelve for B. The formulas are only approximate; and there is no rigorous proof that the restrictions used in Chapter 5 are sufficient, only that they are necessary to prevent a particular type of "worst case" construction. However, if my conjecture is correct, eight middle blocks should be adequate for most programs on Matrix A and twelve should be adequate for most programs on Matrix B.

Matrices C and D represent two possible designs for the internal matrix (which is common to both the small and the large systems). The analog configuration described in Chapter 7 requires 25 matrix inputs and 49 matrix outputs. Applying the formulas of Chapter 5, we find that m=5 and n=3 are the "optimum" values. Since the value of N is not divisible by n and the value of M is not divisible by m, we increase the number of inputs from 25 to 27 and the number of outputs from 49 to 50. This gives Matrix D, which requires 835 switches. This design is presumably optimal for 27 inputs and 50 outputs; but because it has two more matrix inputs and one more matrix output than necessary, it may not be the optimal design for 25 inputs and 49 outputs. Small matrices are generally dominated by divisibility considerations; rounding off a number to the nearest integer makes a much greater difference if the number is small. Matrix C was designed by taking divisibility into account: no extra inputs or outputs were added. The numbers 25 and 49 were simply factored in the only way possible. Note that this gives n=5 and m=7 so that $n<m$, as it should be.

9.2.2 Results of Random Output-full Programs

About 20 randomly generated programs should be enough to determine if a matrix is, in fact, adequate for most programs. Twenty programs were run on Matrix A and 17 on Matrix B. The formulas in Chapter 5 suggest eight middle blocks for Matrix A_9 and eight were adequate in 18 out of 20 cases. The other two out of 20 took 9. There were no cases taking fewer than eight or more than nine. Since these 20 cases were all output-full, which is a "worst-case" condition, it appears that eight middle blocks should be adequate for much more than 90% of all programs.

For Matrix B, the formulas dictate 12 middle blocks. Out of 17 programs run, 16 took 12 and one took 13 middle blocks. Hence 12 middle blocks should be sufficient for the vast majority of programs.

For matrix C and D we want not only to evaluate the efficiency of the matrices and the algorithm, but also to compare the matrices against each other. This requires more than 20 runs for each matrix. Fortunately, these matrices are small, so that they don't take much time.

A total of 90 programs were run on Matrix C. As expected, most of them took seven middle blocks, but three of them took only six and eleven took eight. Out of 50 runs on Matrix D, 45 took five middle blocks and five took six. In both cases, the recommended number of middle blocks (Y=m) is adequate for about 90% of the tested cases. Hence Matrix C is recommended because it uses fewer switches.

In all four cases (Matrices A, B, C, and D) we found the recommended number of matrices adequate about 90% of the time. Should we add an additional middle block to take care of the remaining 10%? It turns out that the answer is no. Remember that output-full programs constitute a "worst-case" set. The more relevant question is whether the proposed matrices are adequate for "typical" programs. This question is taken up in the next section.

9.2.3 Random Programs with Prescribed Fanout Distribution

The results of Section 9.2.2 indicate that the design formulas in Chapter 5 are good ones for "worst case" programs. But how likely is a "worst case" to actually occur? What results are we likely to find for typical, rather than pessimistic, assumptions?

It seems clear that the number of components used and their fanout are the most important parameters in determining the difficulties we are likely to encounter. Hence a routine was written to generate random programs with a prescribed probability distribution.

Let Pi be the probability that a given randomly selected input has a fanout of i. We include the case i=0 (which means that the component is either idle or has internal connections only). Given any sequence $P_0, P_1, P_2, \ldots, P_i$ for which all P_i are non-negative and the sum of the P_i is unity, this routine generates random programs according to this distribution. It works as follows:

For each input a "random" (really pseudo-random) trial with this probability distribution is made. This determines the fanout for that input. Suppose the result is the integer k. Then that component output (matrix input) must connect to k component inputs (matrix outputs). These matrix outputs are selected at random (equal probability for all) from the set of available outputs. Since a matrix output cannot be connected to more than one matrix input, each output that is selected is removed from the list of "available" outputs. Hence as outputs are chosen, the choice is made from a steadily shrinking set.

This process is continued until all inputs are taken care of. Since the number of matrix inputs is fairly large (50 for Matrix A, 100 for Matrix B), the resulting program should have a fanout distribution close to the given one.

It is conceivable that this routine will run out of outputs before every matrix input is taken care of. How likely this is depends upon the expected number of outputs used. The fanouts form a sequence of random variables with a common distribution function. The mean value of this distribution (i.e. the average fanout per input) is

$$\bar{F} = \Sigma i \cdot P_i = P_1 + 2P_2 + 3P_3 + \ldots$$

Since there are N matrix inputs, the expected number of outputs used is $N\bar{F}$. If this is less than the number of matrix outputs M, then the majority of the programs will use less than M inputs; if it is greater, then the majority will use more than M inputs. (This means that, on the average, most programs can be expected to use more outputs than the matrix has, which is impossible.)

Hence we expect the distribution to satisfy the condition

$$\Sigma i P_i \leq M/N = E$$

In other words, the average fanout cannot exceed the expansion factor of the matrix.

Even if this condition is met, the random program generator routine will occasionally generate a program which uses up all the outputs before all inputs are taken care of. Such a program does not represent a valid analog program and should not be included in statistical tabulations. Fortunately, this happens rarely if the average fanout is much less than E.

Fanout distribution data may be obtained by programming an actual problem and counting the components with fanout of 0, 1, 2, etc. For example, the Saturn IV stage control study (phase 3) was programmed in six modules, with a total of 150 component outputs (matrix inputs). Of these inputs

117	had	fanout	of 0
25	"	"	" 1
3	"	"	" 2
1	"	"	" 3
2	"	"	" 4
1	"	"	" 5
1	"	"	" 6
150	total		

This table refers only to external matrix connections. Components with fanout of zero were not necessarily idle, but they did not need connections on the external matrix.

From this, we could calculate the probability of a particular fanout by dividing the number of inputs having that fanout by 150. Note that 117 out of the 150, or 78%, of the inputs had fanout of zero. This is in agreement with the assumption of light traffic that was used in designing the matrix; we allowed for 50 of the 150 inputs to be in use at any one time, and this problem (which uses most of the components in six modules) uses only 33 of them.

However, if we use these probabilities in the actual 150-by-192 matrix, then it may happen by chance that more than five inputs are in use in one input block or more than eight outputs are used in one output block, even though the average number of inputs per input block is only 3.3 and the average number of outputs per output block is only 4.42. We assume (see Section 6.7) that if this happens, we can equalize the input blocks and/or output blocks by interchanging some components. Hence our real interest is in the 50-by-96 matrix (Matrix A). We have

seen that every program on the recommended 150-by-192 matrix is equivalent to a program on Matrix A provided it uses no more than five inputs per input block and eight outputs per output block. To translate such a program into a Matrix A program, all we need to do is delete ten unused inputs from each input block. When we do this, of course, the probabilities change. We still use only 33 inputs, but it is 33 out of 50, not 33 out of 150.

In terms of Matrix A, the distribution looks like this:

17	inputs have fanout of	0
25	" " "	1
3	" " "	2
1	" " "	3
2	" " "	4
1	" " "	5
1	" "	6
50	total inputs	

This gives P_0=0.34; P_1=0.50; P_2=0.06; P_3=0.02; P_4=0.04; P_5=P_6=0.02.

Using this distribution, 20 programs were generated and implemented on Matrix A. Three of them used five middle blocks, eight used six, and nine used seven. Note that none of the 20 required eight middle blocks. The median number of middle blocks used was six; with this many middle blocks, 55% of the programs could have been implemented. The mean number of middle blocks required was 6.3. When the actual Saturn IV, phase 3, program was implemented, it took six middle blocks.

Thus this specific program is typical of programs with this distribution. But how typical is the distribution itself? Several other programs which used most of the analog equipment were examined; in each case, the distribution was about the same, except for the "tail end" of the curve. In other words, the percentage of inputs with fanouts of 0, 1, 2, or 3 was about the same; but some programs had noticeably more high-fanout inputs than others.

Such large fanouts generally tend to fall into one of the patterns covered in Section 6.6; that is, they tend to feed several similar components so that the deliberate grouping of similar components in the same output block allows a considerable amount of output-block fanout, and hence the effectivefanout (on the programming array) is much less than the actual fanout. Of course, what really counts is how many entries there are in the programming array for a given input (i.e. how many output blocks it must be connected to).

As an example, consider the Voyager first-stage ascent problem described in Chapter 8. This program took ten middle blocks (instead of twelve) on the large system external matrix. The fanout distribution was as follows:

Fanout	Number of Inputs	Probability
0	22	.328
1	34	.507
2	3	.045
3	2	.030
4	0	.000
5	1	.015
6	4	.060
9	1	.015

Since this problem used eight modules instead of six, the probabilities were calculated on the basis of 67 matrix inputs instead of 50. (That is, the eight modules contain 200 matrix inputs, and the external matrix is designed for a maximum density of 1/3.) The number of matrix inputs actually used was 45, so that the number of "zero fan-out" inputs was taken as 67-45=22.

When this particular program was implemented on the external matrix, only one matrix input had a "fanout" of four and none of the others had an effective fanout greater than two. (The term "effective fanout" here refers to the number of output blocks to which the input connected, which is simply the number of entries in the programming array for that row.) Since there were six inputs with fanouts greater than four and eight with fanouts greater than two, this result would be extremely surprising if it occurred in a randomly generated matrix. Fanout within an output block is quite rare in randomly generated programs; but in actual problems like this one, it is fairly common because of the matrix design described in Section 6.6. Hence a randomly generated set of programs based on this probability distribution would be misleading as it would not reflect the fact that output/block fanout is much more common than a random selection would suggest.

It would be desirable to modify the routine for generating random programs to increase the probability of output block fanout, at least for the high-fanout variables. A simpler way of achieving approximately the same objectives would be to reduce the large fanouts arbitrarily. Suppose we decrease all fanouts of four or greater by

one (that is, list an input with a fanout of four as if it had a fanout of three, and similarly for higher fanouts). Suppose we also truncate the distribution at a maximum of six; that is, treat all variables with fanouts above six as if the fanout were six. This goal could actually be achieved by using a redundant amplifier to drive some of the fanouts; and, in fact, it may be necessary for electrical buffering anyway. These rules are arbitrary of course, but they seem reasonable and they do not reduce the fanout as much as the deliberate matrix design in Section 6.6 did.

If we apply these rules to the above distribution, the values of P_0, P_1, P_2, and P_3 remain unchanged; but P_4 becomes 0.015, P_5 becomes 0.060, and P_6 becomes 0.015. This makes the distribution extremely close to the Saturn IV, phase 3, distribution.

Figure 9-1 shows the cumulative distribution functions for the above distributions; that is, each entry represents the probability of a fanout less than or equal to the value on the horizontal axis. These values may be obtained from the P_i values by summation. The actual cumulative distribution function is discontinuous (it has jump discontinuities at each integer value on the horizontal axis). However, the trends may be seen more clearly by joining the points with straight lines.

Curve number 1 is the Saturn, phase 3, distribution; curve number 2 is the Voyager, first stage ascent, distribution; and curve number 3 is the result of reducing and truncating the Voyager distribution according to the rules given above. Note that the curves are virtually indistinguishable for fanout values of 0, 1, and 2. For higher fanouts, the curves are noticeably different, but the above rules for reducing large fanouts make then fairly close.

Examination of other problems tends to confirm this distribution; it is fairly typical of problems that use most of the analog equipment within each module. Problems which leave many components idle within a module have less fanout since there is a greater percentage of idle inputs. This is typical of problems which do not have approximately the same balance of components as the computer itself. For example, the heat flux problem submitted by NASA (a partial differential equation) used lots of integrators and very little nonlinear equipment; hence traffic on the matrix was extremely light for this problem.

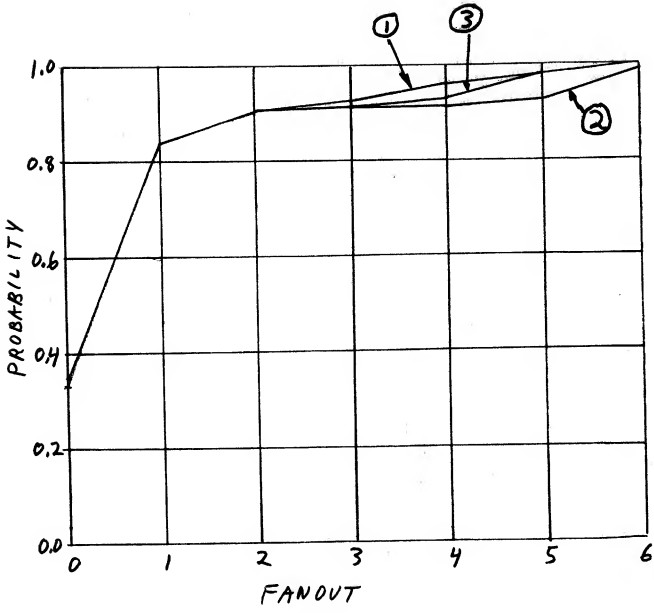


Fig. 9-1. Fanout Probability Distributions.

10 SOFTWARE

This chapter describes the various types of software that are either necessary or desirable for the proposed system. A few of the routines in this chapter have already been written, since they proved necessary for the feasibility study. However, in their present form, they cannot really be considered "software" in the official sense. They lack the detailed documentation and the other backup services of regular software. Of course, they also lack analog hardware with which to operate.

10.1 The Switch Assignment Routine

This algorithm is described in Section 9.1. It accepts as input a programming array, that is, a set of statements of the form "The I-th input on the J-th input block is to be connected to the K-th output block." Note that this is not a complete description of the original program since it does not specify which output or outputs it connects to within the output block, but this is not necessary for a feasibility study. The output is a print-out of the programming array with a middle-block number for each entry.

10.2 The Matrix Terminal Assignment Routine

This routine accepts as input a set of connection statements of the form "Connect input A on component 4-2 (that is, summer number 2 in module 4) to the output of 3-15 (multiplier number 15 in module 3)." The routine also uses as input matrix description data specifying which input and output terminals correspond to these inputs and outputs. This information, which is listed in Tables 7-1 through 7-6, is presently fed in on punched cards. In an operating system, such information would be kept on the system tape.

This routine produces as output a description of the programming array in the proper format for use as input to the switch assignment algorithm.

The first two routines have been written and were used to program the problems in Chapter 8. The following routines have not been written. In some cases the writing of the routine is a straightforward (but tedious) task; in some cases a considerable amount of effort may be required to devise a suitable algorithm.

10.3 Interconnection Routine

This routine, operating in conjunction with the other two, would generate the bit-patterns for actually energizing the switches. This

bit-pattern might be put out on cards or paper tape for later setup (like stored pot-settings) or it might be used to close the switches directly if the GPDC is operating on-line with the GPAC.

These three routines taken together constitute a bare minimum of software for an operating system. This system will be called System A. With System A, the user prepares his circuit diagram in the conventional manner, sections it into modules, and writes connection statements which are punched on cards and used as input. The GPDC takes over from there, either setting up the GPAC directly or providing him with a punched tape or cards to do it. If there are not enough middle blocks in the switching matrix to do the job, then the programming array may be printed out and the user can examine it off-line, and decide if he can relieve the blocking by re-assignment of components.

10.4 A Component Assignment Routine

The next step would be to add a routine for component assignment. Such a routine would accept the same type of input (connection statements) as in System A, but with arbitrary labels. The programmer could call a particular integrator I15 or I22, without regard to the numbering system of the computer. He could even call it XDOT or THETA or some other mnemonic label. Of course he must specify that the component is an integrator.

As pointed out in Section 6.7, such a routine would consist basically of a sectioning subroutine and a subroutine for interchanging components within a module. A general outline of the latter is given in Section 6.7, but the former is quite difficult. I have sectioned enough programs visually to convince myself that it can be done, but the process is an art rather than a science. Much research is needed to get this process into the form of an algorithm.

Originally, I thought that the number of components used and their type would be independent of sectioning. The input was considered as a set of statements about what type of component was connected to what type of other component and was therefore equivalent to a circuit diagram without component assignment. The assignment algorithm was thought of as equivalent to writing numbers inside the triangles and rectangles, that is, deciding which particular component to use. Sectioning the problem is part of this process.

It now appears that this description is too simple. Sectioning also includes re-arranging the triangles and adding or subtracting a few.

Consider, for example, the discussion in Section 6.4 about the various ways to add variables in one module and send the sum to another module. Some methods require fewer inter-module signal paths than others, but perhaps the one that uses fewest signal paths cannot be used because there are not enough summers available in a particular module. Every program that I have sectioned involved decisions of this type, and the final circuit diagram was not topologically equivalent to the original unassigned diagram. The sectioning algorithm should have the freedom to make such changes so that the programmer may feed in a connection statement calling for a three-input integrator and get back a program which uses a three-input summer followed by a one-input integrator or perhaps a two-input summer followed by a two-input integrator. The software has to be capable of doing this in such a way that the signs work out properly; this includes changing the polarity assignment on multipliers or DFG's and shifting the inverters around in linear loops. The bookkeeping problems are by no means trivial.

A system including automatic component assignment will be called System B. System A was described as a bare minimum of operational software; some users may feel that this is too Spartan and insist on B as a minimum system. It may be desirable to define a system between A and B, (System B flat?) which would force the user to section the problem himself but would include the routine for interchanging components within a module. The reason for suggesting this is that the interchanging routine appears to be less of a task to write than the sectioning routine.

10.5 Further Software Capabilities

System B contains all the capability that I (as an analog programmer) would consider necessary to solve the problem of automatic patching (and, hence, cheap and convenient analog stored program capability). However, there are many additional features that are desirable and probably feasible, once we have gone this far. System C (the deluxe system) might contain any or all of the following:

a) The ability to generate the necessary connection statements for a complicated algebraic expression from a FORTRAN statement.

b) The ability to generate the connection statements for a second-order loop, a Padé filter, a backlash circuit, etc. from a single statement.

c) Automatic static check calculations.

d) Automatic scaling.

e) Automatic generation of dynamic check solutions.

f) Diagnostics for checking switch continuity, accuracy of analog components, etc..

g) Character generation for labeling of plots, recordings, scope displays, etc..

Most of Systems A and B can be written in FORTRAN and hence made available on any computer. A fair part of System C may have to be written in assembly language to avoid excessive storage and time requirements. In fact, the capabilities of System C make it look very much like a compiler itself. Creating such a system would be very much like writing a compiler, although there would doubtless be some features unique to this task because of the modular parallel nature of the analog computer.

REFERENCES

1. Clos, Charles. "A Study of Non-Blocking Switching Networks" Bell System Technical Journal, March 1953 Vol 32 pp 406-423

2. Hagerbaumer, William G, and Marshall, Joseph H. "Switching Network Considerations and Configurations in an Automatic Patching Network for the GPAC" Electronic Associates ASG Report 5-61 May 24, 1961

3. Ocker, Wolfgang. "Considerations and Configurations of an Automatic Patching System for the Idea Computer." Electronic Associates ASG Report 7-61 July 13, 1961

4. Allen, D.W. "S-IV Stage Control Study". Problem Number 750152 (furnished by NASA)

5. Brown, Donald R. "Simulation of the Saturn IB/S-IV/Voyager during First-stage Ascent." A&ES Publication 65-750102 Contract NAS8-11209, December 1965 (furnished by NASA).